UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA

Departamento de Arquitectura de Computadores y Automática

TESIS DOCTORAL

Utilización de unidades especulativas en síntesis de alto nivel

Using speculative functional units in high-level synthesis

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

Alberto Antonio del Barrio García

Directores

Mª Carmen Molina Prego
Román Hermida Corea
José Manuel Mendías Cuadros

Madrid, 2011

ISBN: 978-84-695-0991-3

Utilización de Unidades Funcionales Especulativas en Síntesis de Alto Nivel

Using Speculative Functional Units in High-Level Synthesis

Tesis Doctoral

Alberto Antonio Del Barrio García

Departamento de Arquitectura de Computadores y Automática
Facultad de Informática
Universidad Complutense de Madrid

2011

Utilización de Unidades Funcionales Especulativas en Síntesis de Alto Nivel

Using Speculative Functional Units in High-Level Synthesis

Tesis presentada por
Alberto Antonio Del Barrio García

Departamento de Arquitectura de Computadores y Automática
Facultad de Informática
Universidad Complutense de Madrid

2011

Utilización de Unidades Funcionales Especulativas en Síntesis de Alto Nivel

Memoria presentada por Alberto Antonio Del Barrio García para optar al grado de Doctor por la Universidad Complutense de Madrid, realizada bajo la dirección de D^a. M^{a}Carmen Molina Prego, D. José Manuel Mendías Cuadros y D. Román Hermida Correa (Departamento de Arquitectura de Computadores y Automática, Universidad Complutense de Madrid).

Using Speculative Functional Units in High-Level Synthesis

Report presented by Alberto Antonio Del Barrio García to the Complutense University of Madrid in order to apply for the Doctor's degree. This work has been supervised by Ms. M^{a}Carmen Molina Prego, Mr. José Manuel Mendías Cuadros and Mr. Román Hermida Correa (Computers Architecture and Automation Department, Complutense University of Madrid).

Madrid, 2011

Este trabajo ha sido posible gracias a la Comisión Interministerial de Ciencia y Tecnologia, por las ayudas recibidas a través de los proyectos CICYT TIN2005/5619 y CICYT TIN2008/00508

A Adela, Antonio y Ana Rosa
... y también a Sara

Agradecimientos

Para obrar, el que da debe olvidar
pronto, y el que recibe, nunca

Séneca

En la vida hay momentos buenos y momentos malos. Seguramente el protocolo de este momento exige que deje de lado los malos momentos y me acuerde de los buenos porque pesan más, pero no sería justo y estaría mintiendo. Realmente terminar la Tesis justifica esos malos momentos. Además siempre he pensado que el protocolo es un invento mediocre que encorseta al ser humano, nos fijamos más en la forma que en el contenido, y bastantes formalismos tiene ya esta Tesis como para seguir el guión en algo tan subjetivo como los agradecimientos. Por ello, esta Tesis va dedicada principalmente a toda esa gente que me ha apoyado en los momentos malos y que ha depositado su tiempo y confianza en mí en los momentos de trabajo *sucio*, de esfuerzo de creación, de garabatos en papel, de veranos trabajando para mandar al DATE, en fin, de ese tipo de trabajo que no sale en los *papers*.

En primer lugar, he de agradecer especialmente el esfuerzo realizado por Dña. Mª Carmen Molina Prego, directora de esta Tesis, quien ha contribuido a mis inicios como investigador, desde que era becario, y me ha guiado en la escritura de los artículos y demás. Sus aportaciones e incesable interés por la Síntesis de Alto Nivel han sido fundamentales para mejorar mi capacidad crítica e iniciarme en la vida investigadora universitaria.

Debo agradecer también el tiempo dedicado por D. José Manuel Mendías Cuadros, codirector de esta Tesis. Sus comentarios precisos y amplio conocimiento sobre la Síntesis Conductual me han ayudado en la búsqueda de nuevas soluciones.

Tengo que expresar mi más sincero agradecimiento a D. Román Hermida Correa, codirector de esta Tesis. Fue él quien me ofreció la posibilidad de ser becario de colaboración cuando estaba terminando la carrera, y sin su tiempo y valiosos comentarios científicos esto no habría sido posible. Aprendí mucho cuando fue mi profesor de *Arquitectura*, he aprendido mucho de su rigor científico en el desarrollo de esta Tesis, y tengo la sensación de que

seguiré aprendiendo de él.

Estoy profundamente agradecido a la Prof. Seda Ogrenci Memik. Mi estancia en la Northwestern University supuso un punto de inflexión en mi modo de ver las cosas tanto en la vida como en el mero aspecto de la investigación. Las reuniones semanales en el *Tech. Building* de Evanston durante el último trimestre de 2008 fueron claves en el desarrollo de esta Tesis. Siempre recordaré su implicación y cercanía con los estudiantes ... ya sea lunes o domingo, verano o invierno, desde Venecia o desde Egipto. Es un honor trabajar con ella.

I am deeply grateful to Prof. Seda Ogrenci Memik. My stay at the Northwestern University was a turning point in my life and in my researching career. Weekly meetings at the *Tech. Building* in Evanston during the last trimester of 2008 became essential for the development of this Ph.D. Thesis. I will always remember her interest and closeness to the students ... either on Monday or Sunday, either on summer or winter, either from Venice or Egypt. I am honored to work with her.

Agradezco también a D. Francisco Tirado que me animase a publicar el proyecto de *TARDP*, a la postre semilla de esta Tesis y sin la cual todo habría sido diferente.

Por supuesto, muchas gracias a los revisores del DAC'09 y del DATE'10. El merecido *reject* del DAC'09 unido a los valiosos comentarios de los revisores han contribuido sobremanera a esta Tesis. Y si en el DATE'10 el artículo no hubiera sido aceptado, después de haber sido *rebotado* desde el mismo DAC'09 e ICCAD'09 (por ser demasiado abstracto para un congreso de primer nivel), quién sabe dónde habrían caído tantas horas de trabajo.

Tampoco puedo olvidarme de mis compañeros y ex-compañeros de despacho: el 229 y el 347. Ellos me comprenden mejor que nadie. Doy las gracias a Fran por ser un magnífico compañero de prácticas durante la carrera, y de seminario y despacho después de ella, a la vez que amigo. A Pablo y su moto, al humor de Carlitos y su teoría sobre la docencia (una cosa sé ...), a la sabiduría de Miguel y su teoría sobre el ahorro de energía en los gimnasios, a la alegría del infatigable trabajador que es Guillermo, a los *retro-cafés* mañaneros con Poletti, a los *useful Juernes* de David Cuesta, a Antonio y sus dilemas aristotélicos, a Lucas e Íñigo por llegar los primeros, a Juanan y su bata, a Jesús y su teoría evolutiva de las aves rapaces, etc. Muchas gracias a todos por compartir frustraciones y grandes momentos, y si me dejo a alguno le ruego me perdone.

Muchas gracias también a mis incansables compañeros de laboratorio en la Northwestern: Jieyi Long, Song Liu, Min Ni, Brian Leung y Clint Wu.

Special thanks also to my tireless lab-mates at Northwestern: Jieyi Long, Song Liu, Min Ni, Brian Leung and Clint Wu.

Agradezco a D. Jacobo Santamaría por enseñarme que más vale un alumno que piensa a un profesor que corrige. Gracias a D. Joaquín Hernández,

siempre recordaré que *El trabajo y esfuerzo de recordar e revuelve mi estómago* (2.718281828...); y que con saber multiplicar números del 1 al 5 ya es suficiente. Agradezco a D. Antonio Vaquero por recordarme que un informático tiene que saber lo que ocurre desde que sucede una excepción en JAVA hasta el nivel del electrón. Y por supuesto, muchas gracias a D. Ramiro, mi tutor y profesor de Matemáticas, Física y Química en Bachillerato. Gracias a él aprendí que es posible que tus alumnos saquen un 10 en Selectividad dejándoles comer helado en clase. Un auténtico genio como profesor y como persona. Y en general a todos los miembros de la comunidad científica y al profesorado. Especialmente a aquéllos que cumplen con los compromisos aunque para ello tengan que perder algunas horas de sueño, fines de semana, o días de vacaciones. Va por ustedes.

Por último, y por ello más importante, me gustaría dar las gracias y dedicárselo a mi madre, a mi padre, a mi hermana y a mis abuelos. He crecido con ellos, me han educado, apoyado en los malos momentos y desde luego que son quienes más se lo merecen. Y por supuesto a Sara, sin su apoyo y comprensión todo habría sido más dificil. Como diria Julio Verne en la última página de su célebre *Vuelta al Mundo en 80 días* ...

And, forsooth, who would not go round
the world for less ?

Y, en verdad, quién no viajaría alrededor
del mundo por menos ?

Dado que esta Tesis versa sobre las bondades de los errores, ruego a Vuesa Merced que me permita iniciarla de esta bella manera

As this Ph.D. Thesis is about the benefits of failures, I beg Thou to let me begin in this beautiful manner

Hoy en día la mayor parte de la gente muere de una especie de sentido común progresivo, y descubren cuando es demasiado tarde que lo único de lo que uno jamás se arrepiente es de sus propios errores

Oscar Wilde, por medio de Lord Henry Wotton en "El Retrato de Dorian Gray"

Nowadays most people die of a sort of creeping common sense, and discover when it is too late that the only things one never regrets are one's mistakes

Oscar Wilde, via Lord Henry Wotton in "The Picture of Dorian Gray"

Index

List of Figures

List of Tables

List of Algorithms

List of Acronyms

CMOS Complementary Metal Oxide Semiconductor

ASIC Application Specific Integrated Circuit

HLS High-Level Synthesis

ESL Electronic System Level

CAD Computer-Aided Design

ASAP As Soon As Possible

ALAP As Late As Possible

FU Functional Unit

ILP Integer Linear Programming

CSE Common Subexpression Elimination

ISPS Instruction Set Processor Specification

RTL Register Transfer Level

HDL Hardware Description Language

CDFG Control Dataflow Graph

DFG Dataflow Graph

CFG Control-flow Graph

VLSI Very Large Scale Integration

EDA Electronic Design Automation

FPGA Field Programmable Gate Array

IC Integrated Circuit

RCA Ripple Carry Adder

CLA Carry Lookahead Adder

cstep control-step

SFU Speculative Functional Unit

CSEL Carry Select Adder

VLFU Variable Latency Functional Unit

DiffEq Differential Equation

RAW Read After Write

WAR Write After Read

WAW Write After Write

FA Full Adder

HA Half Adder

RCLA Ripple-Block Carry Lookahead Adder

BCLA Block-Carry Lookahead Adders

ESTC Estimated Carry Adder

CLASP Carry Lookahead-like Speculative Adder

PRADD Predictive Adder

ADPCM Adaptive Differential Pulse Code Modulation

1BIPP 1-Bit Input Pattern Predictor

RCM Ripple Carry Multiplier

CSAM Carry Save Multiplier

CSA Carry Save Adder

BWM Baugh-Wooley Multiplier

PRM Predictive Multiplier

DCT Discrete Cosine Transform

CenM Centralized Management

DisM Distributed Management

NEPT Non-speculative Execution Paradigm Theorem

SEPT Speculative Execution Paradigm Theorem

CPI Cycles Per Iteration

2EWF Second Order Elliptic Wave Filter

IDCT Inverse Discrete Cosine Transform

Lattice Lattice Filter

LMS Least Mean Square Filter

CSLU Commit Signals Logic Unit

EGU Enable Generation Unit

FSM Finite State Machine

WARCT WAR Cycles Theorem

HFU Hamming FU

HD Hamming Distance

HR Hamming Register

LRUR Least Recently Used Register

LEA Left-Edge Algorithm

Resumen

La eñe también es gente
María Elena Walsh

En cumplimiento del Artículo 4 de la normativa de la Universidad Complutense de Madrid que regula los estudios universitarios oficiales de postgrado, se presenta a continuación un resumen en español de la presente tesis, que incluye la introducción, objetivos, principales aportaciones y conclusiones del trabajo realizado.

0.1. Introducción

Eratosthenes de Cyrene fue un matemático griego, astrónomo, geógrafo, poeta, músico teórico e incluso un atleta. Fue el inventor de la esfera armilar, propuso su famosa criba para encontrar números primos y contribuyó al desarrollo de la ciencia y la tecnologia en su tiempo. Pero hoy se le conoce principalmente por ser el primer ser humano capaz de calcular el radio de la Tierra y, por tanto, su circunferencia. Corría el año 240 AC, no había máquinas. Pero el intelecto humano siempre ha ido más allá de las dificultades. Eratosthenes abstrajo el problema, y con la ayuda de varias estimaciones y relaciones trigonométricas calculó la circunferencia terráquea con un 1 % de error.

A lo largo de la Historia de la Humanidad ha habido mucha gente que ha contribuido a la evolución de la sociedad. Del mismo modo que Eratosthenes, abstrajeron los problemas de sus respectivos tiempos y crearon nuevas soluciones. El primer coche moderno se inventó en el siglo XIX, el primer computador a principios del XX, y actualmente, en los albores del siglo XXI, la vida no puede concebirse sin ordenadores portátiles o teléfonos móbiles.

Estos dispositivos dominan la tecnología actual, ellos son el principal desafío de nuestro tiempo. Albert Einstein dijo una vez: "Solamente dos cosas son infinitas: el universo y la estupidez humana". No estamos seguros de las dos cosas, pero modestamente creo que olvidó mencionar una más: la ambición humana.

Los usuarios quieren dispositivos electrónicos más potentes, más pequeños y con baterias más duraderas. La aparición de los *Complementary Metal Oxide Semiconductor* (CMOS) ha ayudado a satisfacer estas necesidades. La contínua disminución de la anchura del canal en los dispositivos CMOS ha hecho posible reducir el tiempo de ejecución, el área y el consumo de potencia, a pesar de algunos efectos negativos como el incremento de la densidad de potencia, por ejemplo. Sin embargo, este tipo de solución *tan solo* propone disminuir el tamaño de la tecnología. Actualmente el canal de los dispositivos CMOS está alcanzando unas dimensiones realmente pequeñas, y bajar de 45 nm es una tarea realmente difícil, especialmente en la industria de los *Application Specific Integrated Circuits* (ASICs). Por un lado es un proceso muy costoso, y por otro, es preciso superar varios inconvenientes como el incremento del consumo estático, su efecto exponencial sobre la temperatura y la consecuente degradación de los dispositivos.

Sin embargo, los seres humanos son capaces de satisfacer las necesidades de los usuarios desde niveles de abstracción más altos, sin necesidad de llegar al nivel físico. Estas soluciones más abstractas pueden aplicarse sin condicionar el uso de soluciones complementarias de bajo nivel, mientras que las soluciones de bajo nivel sí que condicionan el uso de ciertas soluciones de más alto nivel. Sirva de ejemplo un arquitecto cuyo objetivo es construir un piso de 100 plantas. Puede pensar en el enorme edificio y entonces escoger el acero para consturir la estructura del mismo. Sin embargo, si piensa primero en usar madera o ladrillos, no podrá construir más que una cabaña, o a lo sumo un bloque de apartamentos.

Por otro lado, las soluciones de alto nivel en ocasiones carecen de cierta información que podría ser muy importante de cara a encontrar las mejores soluciones. Pensemos otra vez en el arquitecto diseñando el mismo rascacielos, pero para que sea construido en San Francisco o en Japón. El edificio deberá ser resistente a los terremotos, así que el arquitecto debería pensar en materiales elásticos para disipar mejor la energía de los mismos.

En el contexto de los ASICs, los ingenieros no tienen que diseñar edificios, afortunadamente, pero sí deben buscar el mejor circuito para satisfacer las necesidades del usuario. Esta Tesis tratará de satisfacer estos requerimientos desde el punto de vista de la *Síntesis de Alto Nivel* (SAN), y al igual que el arquitecto, la solución propuesta tendrá en cuenta algunas características especiales de los *ladrillos* que implementarán la especificación inicial dada por el diseñador. En concreto, las implementaciones convencionales serán aceleradas sin incurrir en un incremento notable del área. Esto sera logrado gracias a la adaptación de las *Unidades Funcionales Especulativas* (UFEs), un nuevo tipo de *Unidades Funcionales de Latencia Variable* (UFLVs), a la Síntesis de Alto Nivel.

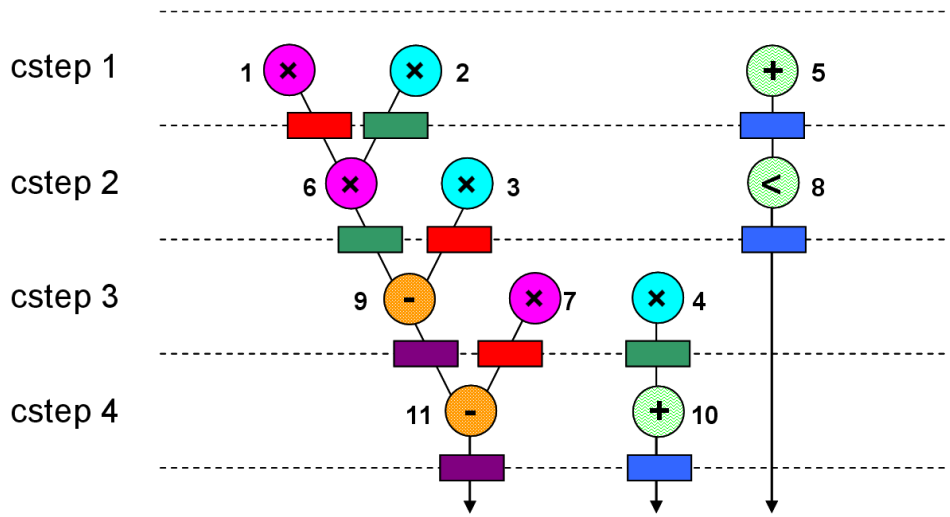


Figura 1: Posible planificación y asignación de UFs y registros en el benchmark DiffEq

cstep	M1	M2	A1	A2	R1	R2	R3	R4
1	1	2	5		1	2	5	
2	6	3	8		3	6	8	
3	7	4		9	7	4		9
4			10	11			10	11

Tabla 1: Asignación de recursos en el benchmark DiffEq

0.1.1. Paradigma de Ejecución No Especulativa

Antes de evaluar detenidamente la aplicación de la especulación a la SAN, en esta subsección se invita al lector a reflexionar sobre qué condiciones son necesarias para escribir las operaciones en los registros de una ruta de datos. Para ello, consideremos el ejemplo de la figura 1, donde se muestran unas posibles planificación y asignación de recursos del benchmark *Differential Equation* (DiffEq). Para aclarar aún más el contenido de esta figura, la tabla 1 muestra un resumen de la asignación de recursos.

Consideremos una ejecución convencional con UFs de latencia fija. Por ejemplo, pensemos en la *Operación 6*. La pregunta es ... cuándo puede escribirse la *Operación 6* en el registro R2? La *Operación 6* puede escribirse en R2 si el controlador está en el estado 2, y si los predecesores de dicha operación han sido escritos previamente. En realidad, comprobar el estado es suficiente para garantizar la segunda condición, por construcción de la planificación, porque estar en estado 2 garantiza haber escrito las *Operaciones*

1 y *2*, correspondientes al estado 1. Sin embargo, esta división de condiciones será necesaria para entender posteriormente las condiciones relativas al paradigma especulativo.

0.1.2. SAN y Especulación

Las Unidades Funcionales Especulativas son Unidades Funcionales de Latencia Variable que predicen ciertos valores intermedios para disminuir el camino crítico de la *Unidad Funcional* (UF) en cuestión. El rendimiento medio dependerá de la tasa de acierto. Varios trabajos [WDH01, ADH05, Cil09, VBI08] han presentado diversos diseños de UFEs, más o menos rápidas, con mayor o menor área, etc. Pero todos ellos tienen en común que siempre existirá la posibilidad de fallo en la predicción. Para resolver esta situación proponen utilizar un ciclo extra para corregir los fallos, pero considerando únicamente el diseño de una UFE. Sin embargo, el objetivo de un diseñador no es hacer UFs sino utilizarlas en un circuito.

Por tanto, *¿qué pasa si hay un fallo en un circuito con varias UFEs?* Ésta es la pregunta que debe ser respondida.

En el trabajo presentado en [Mue99] se propone utilizar un planificador basado en la antigüedad para gestionar varias UFLVs. Sin embargo, el trabajo está desarrollado en el contexto de los procesadores, y la introducción de varias FIFOs supone una penalización demasiado alta desde el punto de vista de la SAN, donde las rutas de datos deben estar muy optimizadas para satisfacer las restricciones dadas por el diseñador.

Por otro lado, algunas técnicas de presíntesis han sido desarrolladas para integrar las UFLVs en las rutas de datos. El trabajo presentado en [RRL00] propone transformar los *Dataflow Graphs* (DFGs) en *Control Dataflow Graphs* (CDFGs), y tratar las operaciones ejecutadas en una UFLV como una estructura de control de tipo bifurcación. Esto produce una penalización de área considerable, ya que utilizando varias UFLVs el número de combinaciones a controlar es exponencial. Además las UFLVs consideradas en [RRL00] son significativamente más grandes que las correspondientes UFs de latencia fija.

Por otro lado, la introducción de las *Unidades Telescópicas* en [BMPM98] proporciona un paradigma para construir automáticamente circuitos de latencia variable. Cada uno de estos circuitos debe calcular una función de error, llamada *hold function*, que indicará al controlador si el resultado está disponible o no. Por último, en el trabajo presentado en [BCK09] se propone generar circuitos de latencia variable reduciendo el camino crítico mediante la búsqueda de *puntos especulativos* en la netlist del circuito. Un punto especulativo es aquel en el que el uso de la especulación puede disminuir su retardo. Pero estas aproximaciones padecen el mismo problema que [RRL00]: si se utilizan varias UFLVs o puntos especulativos, el número de casos a controlar será exponencial.

Además de la complicación de controlar todos los casos, cuando hay varias UFEs en la misma ruta de datos, la probabilidad de que todas trabajen en modo de baja latencia disminuye. Por ejemplo, supongamos que una determinada UFE tiene una tasa de acierto del 90 %. Si ahora suponemos un circuito con 6 UFEs, con la misma tasa de acierto, la probabilidad de que todas trabajen en modo de baja latencia será de $(0.9)^6$, es decir, un 53 % aproximadamente. Por tanto, el uso de muchas UFEs afectará al rendimiento total del circuito.

0.1.3. Objetivos de la Tesis

Actualmente, la automatización del proceso de diseño se ha convertido en una tarea esencial, debido a la cada vez mayor complejidad de los diseños y al menor tiempo destinado a ellos por parte de las compañías. En este proceso de automatización, tradicionalmente se han utilizado módulos de latencia fija para implementar los circuitos. Así pues, todo el flujo de la Síntesis de Alto Nivel se ha fundamentado sobre esta idea. La aparición de las Unidades Funcionales de Latencia Variable ha permitido romper con este axioma. Sin embargo, todos los trabajos previos han seguido direcciones que no permiten la utilización óptima de este tipo de UFs. Por un lado se han diseñado Unidades Funcionales muy rápidas que trabajan la mayor parte del tiempo en el modo de baja latencia [WDH01, ADH05, Cil09, VBI08], pero es preciso disponer de un control que permita usarlas. Y aunque por otro lado algunos trabajos han conseguido automatizar el desarrollo de este control, solo se ha conseguido considerando muy pocas Unidades Funcionales de Latencia Variable [BMPM98, RRL00].

Las Unidades Funcionales Especulativas son un subconjunto de UFLVs que operan prediciendo ciertos valores intermedios. Si la predicción es cierta, la latencia del módulo será menor. En concreto, las UFEs que serán utilizadas en esta Tesis predicen la señal de carry para disminuir el camino crítico de las UFs. El rendimiento de dichas UFEs será determinado por la tasa de acierto del predictor. Sin embargo siempre existirá la posibilidad de fallo. En este caso, las UFEs necesitan estar coordinadas por algún mecanismo para proceder a las correcciones pertinentes y mantener así la integridad de la ruta de datos.

La razón principal para usar este tipo de UFs es su gran balance entre rendimiento y área/potencia. Tradicionalmente, la SAN trata de satisfacer las restricciones de rendimiento utilizando módulos más rápidos, como los *Carry Lookahead Adders* (CLAs) o los *Carry Select Adders* (CSELs). Sin embargo, su penalización en área/potencia puede producir una violación de las restricciones sobre estos parámetros. Las UFEs reducen el camino crítico pero sin incrementar mucho el área. Por ejemplo, los sumadores propuestos en esta Tesis obtienen un rendimiento similar a los CSELs, pero con un área similar a los *Ripple Carry Adders* (RCAs). Sin embargo, el objetivo principal

de esta Tesis no será el diseño de las UFEs, sino su control. Ya que la mejor característica de las UFEs es el buen balance rendimiento vs área/potencia, la lógica adicional empleada para controlar los errores de predicción debe ser lo suficientemente compleja como para mantener la ruta de datos en un estado correcto, pero lo suficientemente simple como para que el uso de las UFEs sea rentable.

Por tanto, desarrollar un mecanismo de control para corregir los fallos en las predicciones sin impactar negativamente en el rendimiento y sin incurrir en una penalización de área/potencia significativa es el desafío más importante. En esta Tesis se presentan las técnicas necesarias para generar automáticamente un controlador de la ruta de datos que permite la utilización de UFEs en la SAN. Además, varios teoremas y lemas que prueban la corrección de las técnicas propuestas serán formulados y demostrados, y una arquitectura tipo será presentada. Finalmente, después de integrar las UFEs en el flujo de diseño de la SAN, varias técnicas de síntesis serán adaptadas para sacar el máximo beneficio a las características especiales de este nuevo tipo de implementaciones, mejorando el rendimiento de los circuitos sin penalizar en área.

0.2. Diseño de Unidades Funcionales

Las Unidades Funcionales Especulativas son un tipo de Unidades Funcionales de Latencia Variable que predicen los valores de ciertas señales intermedias. En esta Tesis se presentan tanto diseños de sumadores como de multiplicadores predictivos, que obtienen una disminución del tiempo de ejecución del 50 % y del 25 %, respectivamente, en comparación con los correspondientes módulos de latencia fija.

0.2.1. Sumadores Predictivos

La suma es la operación aritmética clave en la mayoría de los circuitos digitales [Kor02]. Por tanto, su rendimiento y otros parámetros, como área, consumo, etc., dependen mucho de las características de los sumadores.

La implementación más directa de un sumador paralelo de dos operandos de n bits es el *Ripple Carry Adder* (RCA), que consiste en la replicación de n celdas básicas denominadas *Full Adders* (FAs). Estas n celdas básicas están interconectadas por la señal de *carry*, lo que determinará el camino crítico del RCA. En resumen, el RCA es una estructura simple, que ocupa y consume poco, pero con un rendimiento pobre.

Para mejorar el rendimiento de los RCAs, tradicionalmente se han utilizado módulos más complejos como los *Carry Select Adders* (CSELs) o los *Carry Lookahead Adders* (CLAs). Los CSELs replican la o las partes más significativas del sumador, ejecutándolas con ambos carrys de entrada '0' y

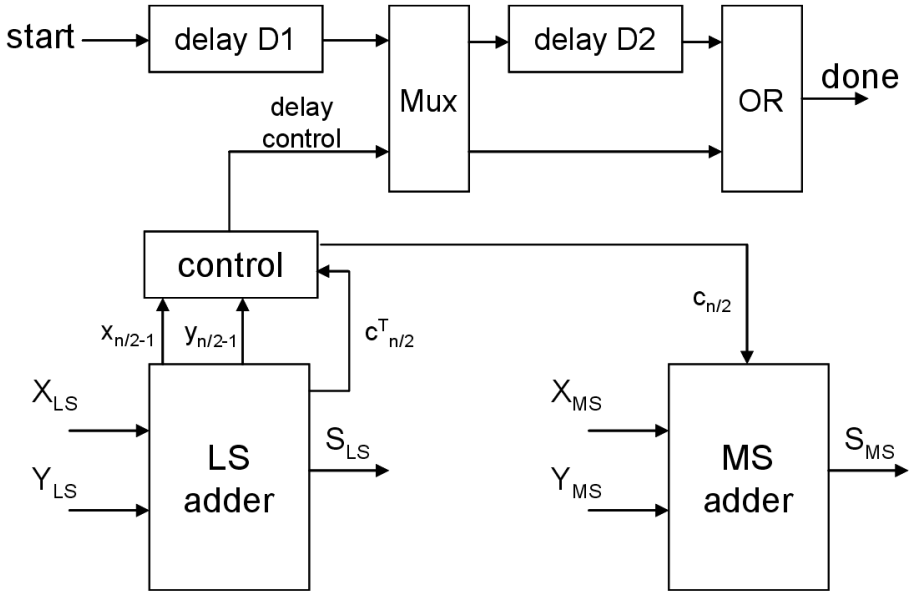


Figura 2: Estructura de un ESTC de n bits

'1'. Los CLAs, por su parte, anticipan los valores de los carrys de entrada, usando uno o varios niveles de anticipación, formando grandes estructuras de árbol en los casos más extremos. De este modo se disminuye el tiempo de ejecución, pero a cambio de un incremento en área bastante considerable.

Aparte de estos sumadores más tradicionales, la literatura nos ofrece diseños más modernos como los *Estimated Carry Adders* (ESTCs) [WDH01, ADH05]. Se trata de sumadores especulativos asíncronos cuya idea principal es implementar un Carry Select Adder sin replicar el módulo más significativo. El carry de entrada a dicho módulo será anticipado por una función combinacional de los bits más significativos de la parte menos significativa del sumador, tal y como muestra la figura 2. Típicamente se utiliza la **AND** lógica de dichos bits, aunque hay opciones más complejas que tienen en cuenta más bits del fragmento menos significativo [ADH05]. Una vez que haya terminado la ejecución del fragmento menos significativo, el valor estimado del carry de entrada al fragmento más significativo es comparado con el carry de salida real del fragmento menos significativo. Si son iguales, la operación termina y se disparará la señal de *done* seleccionando el camino corto del multiplexor de la figura 2. En caso de fallo, se seleccionará el camino marcado como *delay D2* y se cambiará el valor del carry de entrada al módulo más significativo, que efectuará la suma una vez más.

En resumen, el retardo del sumador será el mismo que el de un CSEL en caso de acierto, y aproximadamente el de un RCA en caso de fallo. Sin embargo, a pesar de no replicar el fragmento más significativo, como los

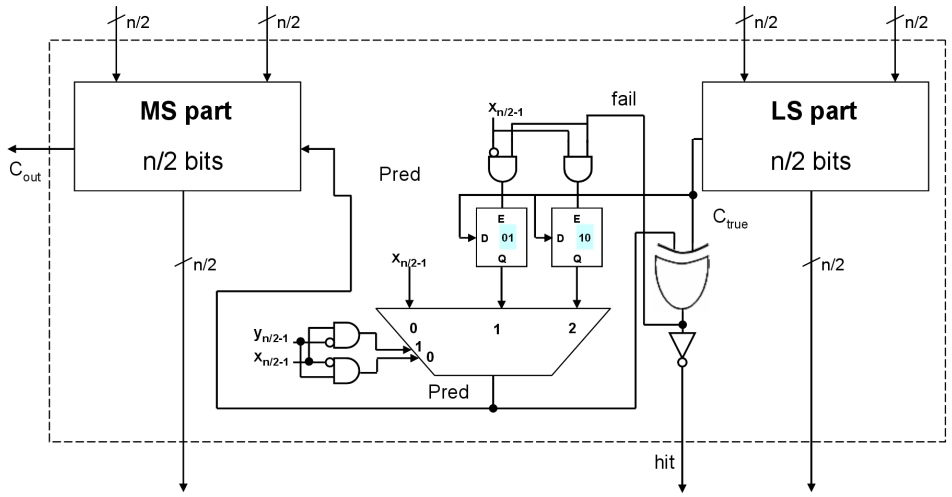


Figura 3: Estructura de un PRADD de n bits con un 1-Bit Input Pattern Predictor

CSELs, los ESTCs tienen dos problemas: por un lado la presencia de bastantes elementos de control y la necesidad de generar una señal de *start*, y por otro lado su asincronismo. Como la mayoría de los circuitos actuales son síncronos, esta asincronía supone un gran problema para poder utilizarlos en el diseño VLSI.

En esta Tesis se propone el uso de los *Predictive Adders* (PRADDs) para vencer la limitación impuesta por la asincronía de los ESTCs, así como para eliminar los elementos de retardo y la generación de señales no útiles. Estos sumadores están divididos en dos fragmentos de igual longitud y tipo de implementación, al igual que los ESTCs. El carry de entrada del fragmento más significativo vendrá dado por un predictor similar a ciertas estructuras presentes en los predictores de salto de los procesadores [HP07]. Por ejemplo, un predictor de 1 bit, el cual puede implementarse con un biestable D. Además, los PRADDs generarán una señal de *hit* para indicar al controlador de la ruta de datos si la predicción es correcta o no.

El funcionamiento de los PRADDs será similar al de los ESTCs, pero en modo síncrono. Es decir, si hay un acierto en la predicción, la suma tardará lo que tarden los fragmentos (un ciclo corto), y si hay un fallo el doble (dos ciclos cortos). Si suponemos que ambos fragmentos están implementados por RCAs, el retardo pasará de n a $n/2$ FAs. Por tanto, el tiempo de ejecución, en caso de acierto, quedará reducido en un 50 %.

La figura 3 muestra la implementación de un PRADD cuyo predictor es un *1-Bit Input Pattern Predictor* (1BIPP). El 1BIPP es en una *tabla* accedida con los bits más significativos de los operandos de entrada al fragmento menos significativo; por tanto tiene 4 entradas. Sin embargo, cuando ambos

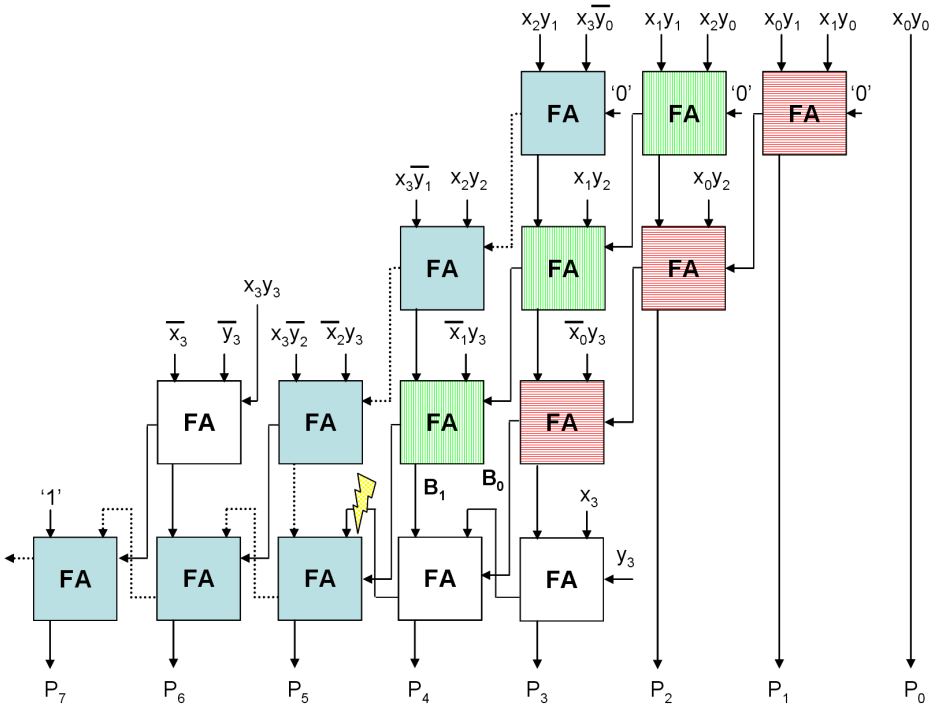


Figura 4: Estructura de un Predictive Multiplier

bits son '0' ó '1', el carry puede ser anticipado porque será igual que dichos bits, por lo que la *tabla* tendrá únicamente dos entradas. Además, para gestionar el acceso y/o escritura del predictor, cierta lógica será necesaria. Nótese la presencia de la señal de *hit*, que valdrá '1' cuando el carry de salida real de la parte menos significativa sea igual que la predicción del carry de entrada del fragmento más significativo.

0.2.2. Multiplicadores Predictivos

Las multiplicaciones son una de las operaciones más comunes en los diseños actuales, por tanto un diseño eficiente de multiplicador es muy importante para satisfacer las restricciones impuestas por el diseñador.

En esta Tesis se propone utilizar una estructura tipo *Carry Save Adder* (CSA) con una última etapa implementada con un Predictive Adder. Como el objetivo es manejar números negativos también, la implementación consistirá en un *Baugh-Wooley Multiplier* (BWM) con un PRADD en la última etapa. Es decir, solamente es necesario usar un predictor por multiplicador. Nótese que el uso de predictores en un multiplicador tipo *Ripple Carry Multiplier* (RCM) implicaría la introducción de un predictor en cada fila del mismo, con las consiguientes penalizaciones de área y rendimiento (ya que el acierto

FU	Delay	Delay Pred	%Pen	Area	Area Pred	%Pen
8-bits RCA	3486	2050	-41.19	6564	7332	11.70
16-bits RCA	6931	3773	-45.56	12966	13619	5.04
32-bits RCA	13822	7218	-47.78	25376	26421	4.12
8x8 BWM	8385	6890	-17.83	43371	44004	1.46
16x16 BWM	17790	14026	-21.16	175511	176445	0.53
32x32 BWM	36833	28523	-22.56	683898	685103	0.17

Tabla 2: Retardo y área de un RCA y un BWM con y sin técnicas de predicción

se produciría solo si todos los predictores acertasen).

La figura 4 muestra la estructura de un *Predictive Multiplier* (PRM). Los bloques coloreados de azul indican el camino crítico del PRM. Si se compara con un multiplicador común mxn, e.g. un RCM, dicho camino pasa de $m+n$ a $\lceil m/2 \rceil + n$ FAs. Así, si $m \approx n$ el tiempo de ejecución quedará reducido en un 25 %. Los bloques rayados indican el número de niveles necesarios para generar los bits B_1 y B_0, cuyo uso será explicado en detalle en el capítulo 2 de esta Tesis.

0.2.3. Resultados Experimentales

Para medir el impacto del uso de predictores en los sumadores y multiplicadores, varios Ripple Carry Adders y Baugh-Wooley Multipliers de diferente tamaño han sido sintetizados con la herramienta comercial *Synopsys Design Compiler*. Los datos están resumidos en la tabla 2. El retardo ha sido medido en picosegundos y el área en μm^2. Tal y como puede observarse, el incremento en área es insignificante, especialmente en el caso de los multiplicadores. En cuanto al retardo, los RCAs predictivos lo reducen en más de un 41 % y los BWMs predictivos en más de un 17.8 %, con respecto a las correspondientes implementaciones no especulativas. Nótese que tanto la disminución en el retardo como la penalización por área reducen su importancia conforme el tamaño de los módulos aumenta. De este modo, cuanto más grande es el módulo, más cerca se está de la reducción teórica máxima de retardo (50 % y 25 % para RCAs y BWMs, respectivamente).

En cuanto a la tasa de acierto de los predictores, el lector podrá encontrar numerosos experimentos en el capítulo 2 de la Tesis, donde comprobará que siempre oscila en torno al 85 %-90 % al menos.

0.3. Control Centralizado de UFEs en SAN

El Control Centralizado es el método más directo para utilizar varias Unidades Funcionales Especulativas en el flujo de la Síntesis de Alto Nivel. La respuesta a la pregunta: *qué pasa si hay algún fallo?* se convierte con esta técnica en una contestación tan trivial como detener la ruta de datos cada vez que lo haya. En cada *control-step* (cstep) habrá que generar una señal global de acierto que indique si todas las operaciones activas en ese cstep han acertado en la predicción o no, para informar al contolador de si puede pasar al siguiente estado.

0.3.1. Fundamentos del Control Centralizado

El Control Centralizado se fundamenta en respetar el Paradigma de Ejecución No Especulativa, que es el paradigma de ejecución convencional, es decir, todo ocurre según la planificación y la asignación, efectuadas de forma estática. Parando toda la ruta de datos cada vez que ocurra un evento anómalo, se garantiza que se respetará la planificación inicial. Dicho paradigma queda resumido en el siguiente teorema:

Teorema 1 (Teorema del Paradigma de Ejecución No Especulativa)**.** *Sean O una operación, SC_o el cstep/estado en el que O ha sido estáticamente planificada, y FU_o y R_o la Unidad Funcional y el registro donde O ha sido asignada, respectivamente. O puede ser finalizada, i.e. escrita en R_o, sii:*

(1) El estado del controlador global es SC_o.

(2) Las dependencias Read After Write, Write After Read, and Write After Write de O están resueltas

Este teorema será demostrado en el capítulo 3 de la Tesis, pero intuitivamente quiere decir que una operación puede ser escrita si efectivamente el controlador se encuentra en el estado en el que fue planificada la operación, y si las dependencias han sido resueltas. Esta última condición es trivialmente cierta, por construcción de la planificación y la asignación de operadores, ya que todas las operaciones se ejecutarán conforme a la planificación inicial. No obstante es interesante mantener esta división de condiciones para establecer similitudes con el Paradigma de Ejecución Especulativa, que será definido más adelante.

0.3.2. Arquitectura del Control Centralizado

La función de la arquitectura del Control Centralizado, mostrada en la figura 5, consiste en generar el siguiente estado y las señales de carga de registros y de enrutado de la ruta de datos, teniendo en cuenta si ha habido un acierto en todas las predicciones o no. De esta forma, se repetirá el estado

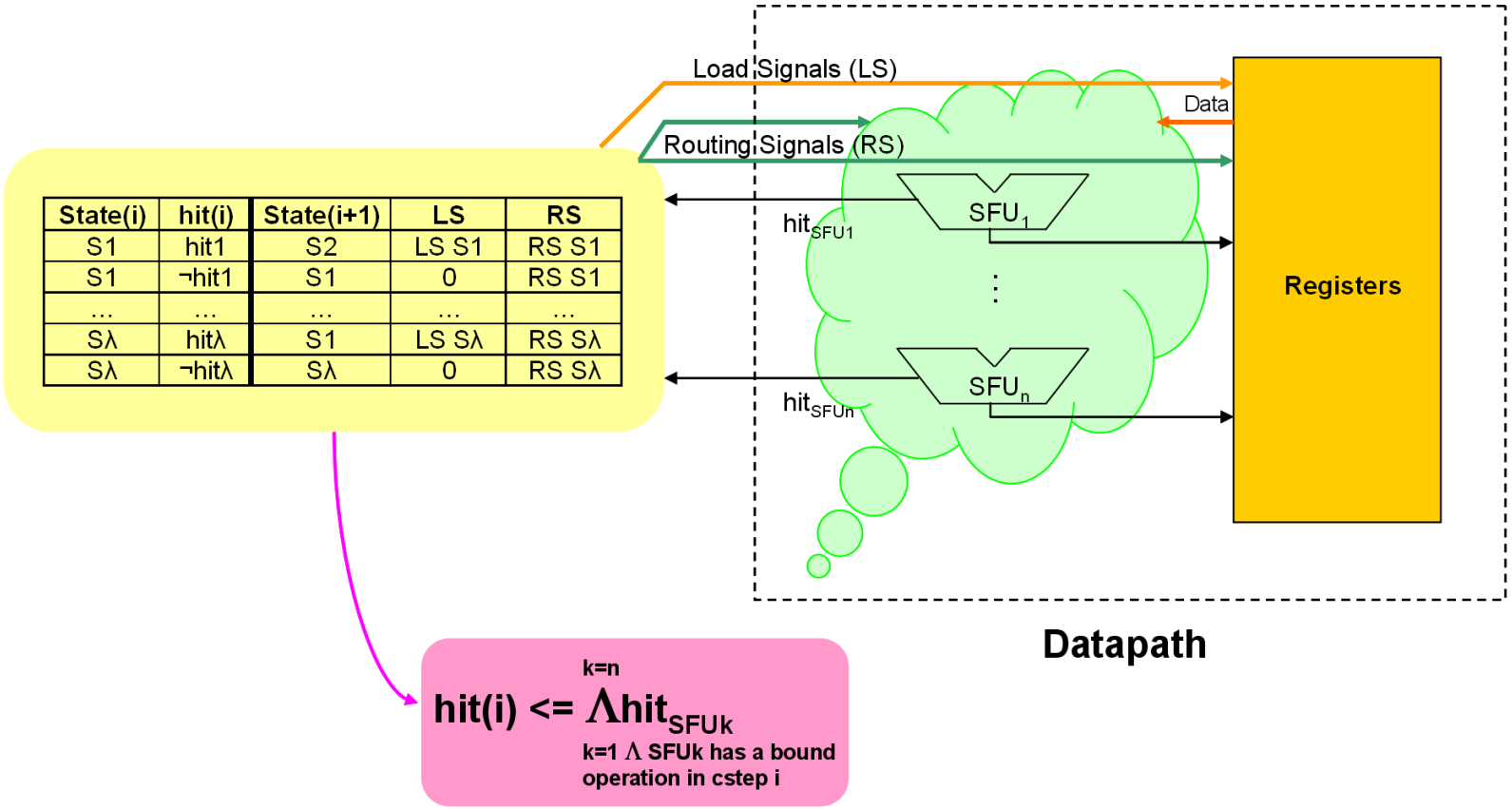


Figura 5: Arquitectura del Control Centralizado

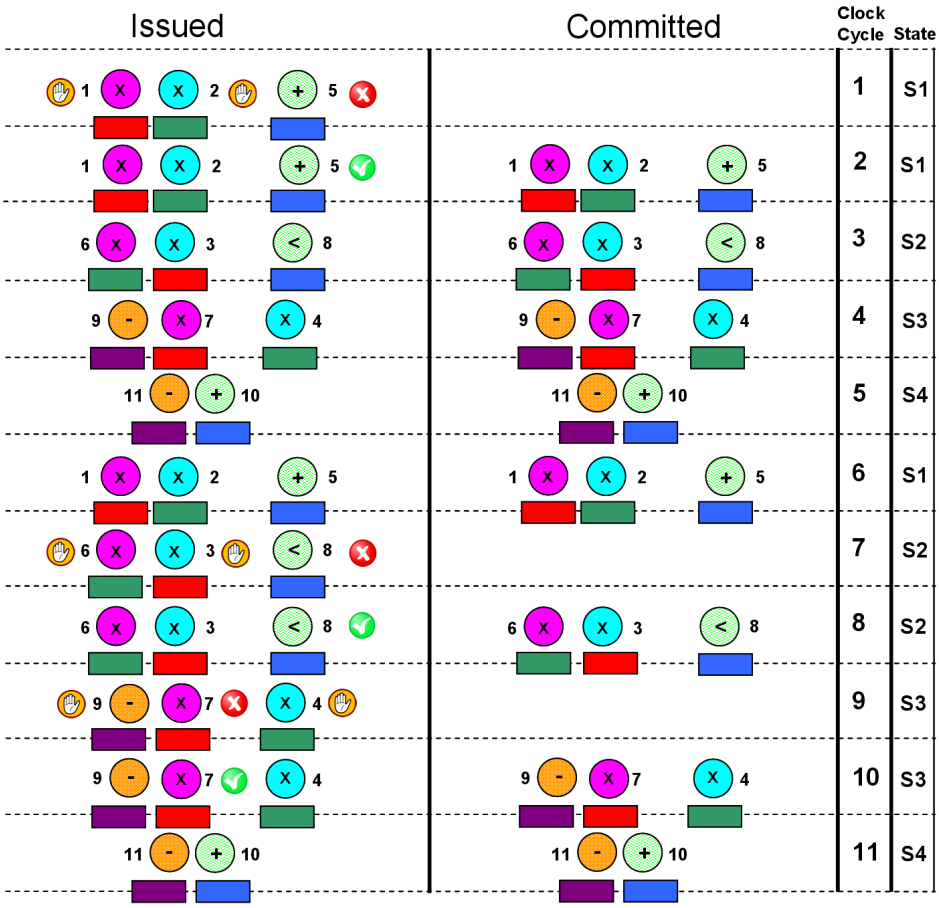


Figura 6: Ejecución de 2 iteraciones del benchmark DiffEq con UFEs monociclo y Control Centralizado

del controlador si se produce un fallo, ya que es preciso cumplir con las condiciones impuestas por el Teorema de Ejecución No Especulativa.

La señal de hit (*hit(i)*) de cada cstep será implementada como una **AND** de las señales de hit de las UFEs activas en dicho cstep. Por ejemplo, en el cstep 2 de la figura 1, la señal de hit será la **AND** de las señales de hit de los multiplicadores M1 y M2, y del sumador A1, ya que son las UFEs que están ejecutando operaciones en el cstep 2.

0.3.3. Ejemplo de aplicación del Control Centralizado

La figura 6 es un ejemplo del funcionamiento del Control Centralizado en el benchmark DiffEq, cuyas planificación y asignación fueron dadas previamente en la figura 1. La figura muestra 4 columnas: las dos primeras

contienen las operaciones que han resuelto sus dependencias de tipo RAW (*Issued*) y las operaciones que pueden ser escritas en su registro correspondiente (*Committed*), respectivamente. Las columnas de la derecha muestran el ciclo de reloj y el estado del controlador. Los símbolos ✘ y ✔ indicarán cuando una operación sufre un fallo de predicción y cuando corrige su resultado, respectivamente. Una mano dentro de un círculo indicará aquellas operaciones que están esperando.

Tal y como puede observarse en la figura 6, la *Operación 5* falla en la predicción en el ciclo 1. Por tanto, las *Operaciones 1* y *2* son detenidas también, y ninguna operación es escrita en el ciclo 1. A continuación, en el ciclo 2, tiene lugar la corrección de la *Operación 5*, y como todas las operaciones activas en el estado S1 aciertan, se produce la transición de estado y la escritura de los registros correspondientes. En los ciclos 7 y 9 se producen dos fallos más, en los que se procede de igual manera. Así pues, para ejecutar 2 iteraciones del DiffEq, con los 3 fallos supuestos, serán necesarios 11 ciclos, en comparación con los 8 ciclos que tardaría una ejecución convencional, sin UFEs.

Pero aparte del número de ciclos, es preciso tener en cuenta que el tiempo de ciclo será menor en el caso de las UFEs. En concreto, teniendo en cuenta los análisis presentados en la sección 0.2, y que normalmente los multiplicadores son los módulos más grandes en las rutas de datos, supondremos que el tiempo de ciclo en el caso de las UFEs es un 75 % del tiempo de ciclo con UFs no especulativas. Por tanto, el tiempo de ejecución del caso común sera de 8 ciclos * 1 unidad de tiempo/ciclo = 8 unidades de tiempo, mientras que el tiempo de ejecución de la implementación con UFEs y Control Centralizado será de 11 * 0.75 = 8.25 unidades de tiempo. Es decir, parar todas las operaciones si tan solo una sufre un fallo, supone una penalización importante, que puede mermar el rendimiento de los circuitos con UFEs. Es necesario buscar un mecanismo de gestión más eficiente.

0.4. Control Distribuido de UFEs en SAN

Es posible ir más allá del Control Centralizado ? Parar todas las operaciones cada vez que haya un fallo es realmente necesario ? En esta sección el lector encontrará respuesta a estas preguntas, así como los conceptos y estructuras fundamentales de una técnica de gestión mejorada para trabajar con UFEs.

La principal idea destrás del Control Distribuido es permitir que las operaciones que no sufren fallos de predicción puedan continuar su ejecución, siempre que los fallos no afecten a las mismas. Sin embargo, permitiendo esto aparecerán nuevos problemas. El primero y más evidente de ellos es el mantenimiento del estado. Si hay un fallo en una UFE y no se detienen todas las demás también, habrá algunas UFEs trabajando en estados diferentes en

un determinado instante. Por tanto, será preciso dividir el controlador global en varios controladores locales, uno por UF. Además, como cada operación puede ser ejecutada independientemente de las demás, será necesario utilizar un estado por operación, en lugar de un estado por cstep.

Al igual que en los procesadores superescalares, permitir que las operaciones se ejecuten siempre que sea posible producirá la aparición de riesgos de tipo *Read After Write* (RAW), *Write After Read* (WAR) y *Write After Write* (WAW). Sin embargo, en el contexto de la SAN no es posible utilizar técnicas de *shelving*, *estaciones de reserva* o *reorder buffers* para mantener el comportamiento correcto del programa. Dado que los circuitos resultantes de la SAN deben ser optimizados al máximo, no es posible utilizar estas estructuras por su alta penalización en área. En su lugar, el Control Distribuido explotará el conocimiento previo del programa, es decir, el *Dataflow Graph* (DFG).

0.4.1. Fundamentos del Control Distribuido

En esta sección será postulado el teorema sobre el cual se sustenta la ejecución de las operaciones con Control Distribuido. Tal y como comprobará el lector, es en cierto modo similar al Teorema del Paradigma de Ejecución No Especulativa.

Teorema 2 (Teorema del Paradigma de Ejecución Especulativa)**.** *Sea O una operación en ejecución. Sean SFU_o y R_o la UFE y el registro donde O ha sido asignada, respectivamente. Sea St_{SFUo} la variable de estado de SFU_o y sea S_o el estado asociado a O. Entonces, O puede ser finalizada, i.e. escrita en R_o, sii*

(1) La predicción del carry de SFU_o es correcta

(2) St_{SFUo}=S_o

(3) Las dependencias Read After Write, Write After Read and Write After Write de O están resueltas

La demostración de este teorema se realizará en la sección 4.1.1 de esta Tesis. Pero como puede observarse, cada operación podrá finalizar si el controlador de la UFE donde fue asignada se encuentra en el estado correspondiente (condición 2), si las dependencias han sido resueltas (condición 3) y si la predicción acierta (condición 1). La condición 1 es inherente al diseño de las UFEs. Pero, las condiciones 2 y 3 son similares a las condiciones del Paradigma No Especulativo, aunque con la diferencia de que el estado que debe comprobarse es el de la UFE donde se ha asignado la operación, y que ahora las dependencias deben ser chequeadas dinámicamente, ya que la planificación cambiará en tiempo de ejecución.

Este chequeo dinámico de dependencias constituye uno de los mayores desafíos de esta Tesis. Debe hacerse en tiempo de ejecución y con una penalización de hardware mínima. En concreto, se utilizarán los estados locales de cada controlador. Para garantizar que una dependencia está resuelta, será suficiente con comprobar el estado de la UFE en la que está asignada la operación causante de la dependencia. Por ejemplo, si tomamos como referencia la planificación y asignación de la figura 1, veremos que el multiplicador M1 tiene asignadas las *Operaciones 1*, *6* y *7*. Por tanto, su controlador puede estar en los estados S1, S6 y S7. Análogamente para el multiplicador M2. Entonces, supongamos que la *Operación 6* está siendo ejecutada. La dependencia RAW con la *Operación 2* debe ser evaluada, entre otras. Si el estado del controlador de M2 es posterior a S2, sabremos con certeza que la *Operación 2* ha sido finalizada y, por tanto, la dependencia resuelta. Además de este mecanismo, un algoritmo para limitar el número de estados a chequear será presentado en la sección 4.1.2.1 de esta Tesis, para disminuir la penalización de área provocada por las condiciones de chequeo.

0.4.2. Arquitectura del Control Distribuido

La arquitectura del Control Distribuido se muestra en la figura 7. Su principal misión consiste en verificar dinámicamente el cumplimiento del Teorema de Ejecución Especulativa. Aparte de ello, esta arquitectura se encargará de generar las señales que gobiernan la ruta de datos.

Tal y como puede observarse, la arquitectura de la figura 7 está compuesta por varios controladores locales, una unidad central de coordinación y por la ruta de datos. Esta unidad central de coordinación se encarga del chequeo dinámico de dependencias, es decir, de disparar las transiciones en los controladores, por medio de la generación de unas señales de enable e_{SFUi}. Por tanto, está dividida en varias Unidades de Generación de Enable (UGE), una por UFE.

La figura 8 muestra una visión más detallada de la arquitectura tipo propuesta para el diseño de UGEs y para generar las señales de rutado y carga de registros.

La figura 8a mapea en hardware el Teorema de Ejecución Especulativa. La señal e_{SFUi} es la **AND** de dos señales: la señal de hit hit_{SFUi} (condición 1), y la señal de salida de un multiplexor. Este multiplexor llevará a su salida una línea de dependencias (condición 3), dependiendo del estado local St_{SFUi} (condición 2). Cada línea de dependencias está compuesta por un conjunto de líneas de salida de multiplexores que resuelven las dependencias. Cada uno de estos multiplexores representa los estados de las UFEs donde están asignadas las operaciones causantes de la dependencia.

La figura 8b muestra la generación del control de los registros. Por un lado la señal de carga $load_{Rm}$. Un registro se cargará siempre que una operación

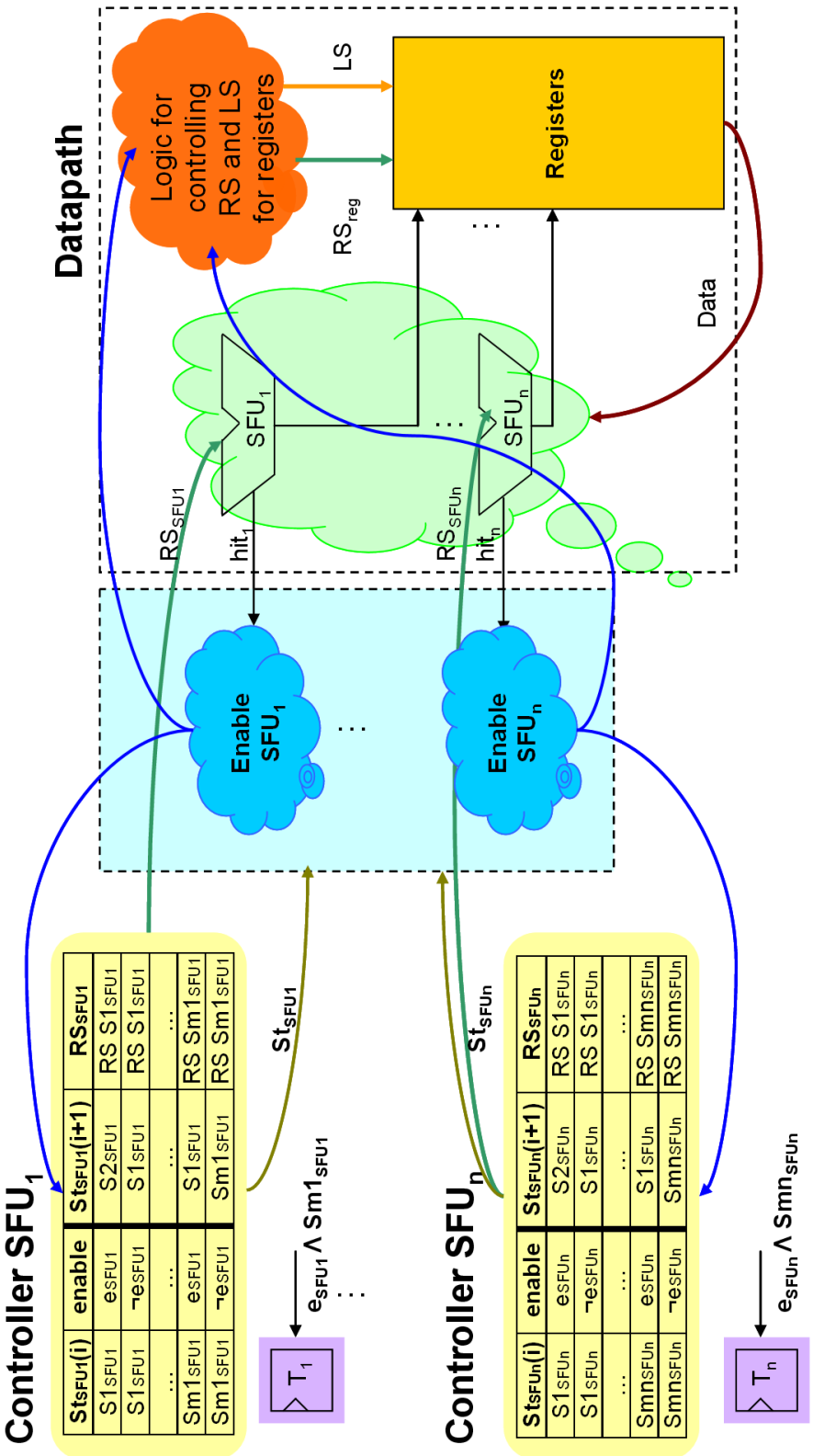
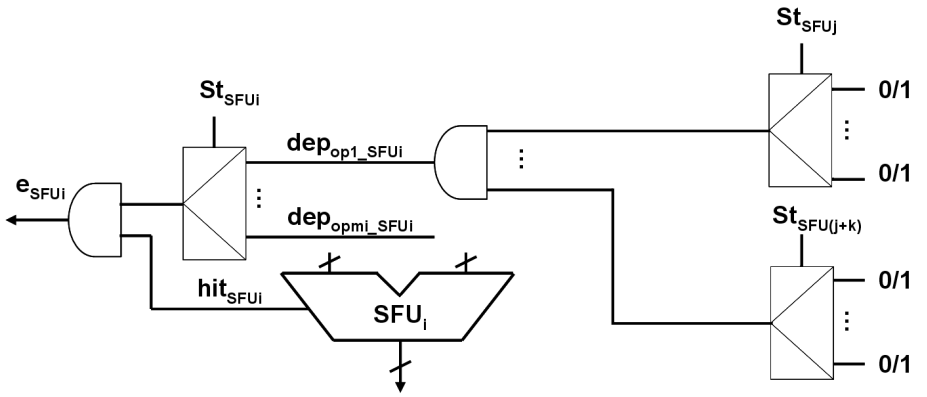
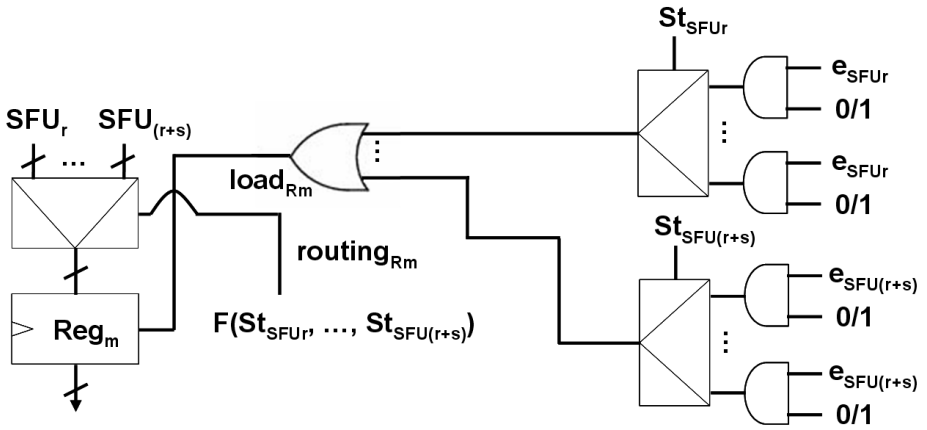


Figura 7: Arquitectura del Control Distribuido



(a) Unidad de Generación de Enable

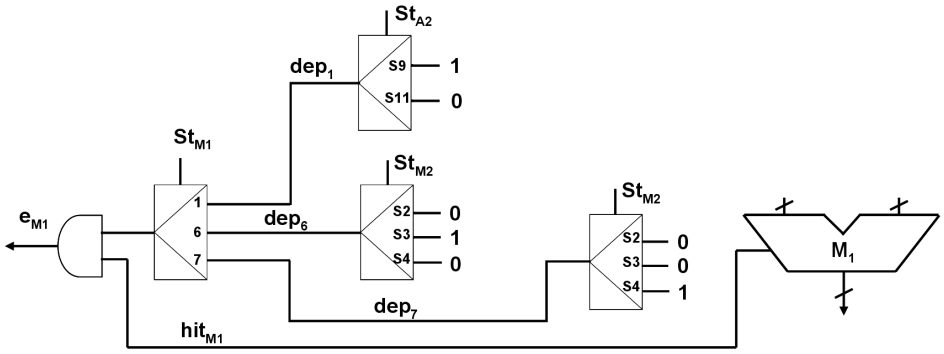


(b) Generación de las señales de rutado y carga de registros

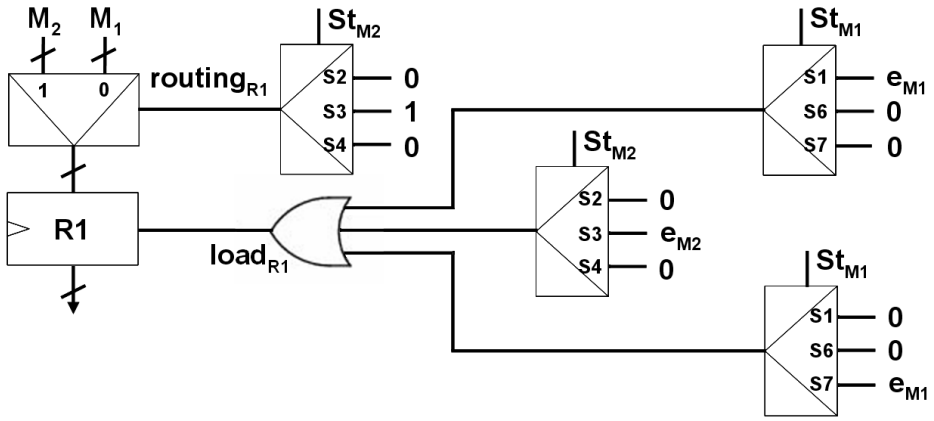
Figura 8: Arquitectura canónica de las UGEs y de las señales de rutado y carga de registros

asignada a él finalice, es decir, se active el enable de la UFE correspondiente y su controlador se encuentre en el estado adecuado. Para seleccionar qué valor cargar en el registro, será suficiente con utilizar un multiplexor e implementar su señal de control en función de los estados locales.

Tal y como puede observarse, las figuras 8a y 8b contienen varios multiplexores con entradas constantes. Nótese que los multiplexores son simplemente una forma de representar los estados, ya que al sintetizar los circuitos la lógica quedará mucho más reducida debido a las constantes. De hecho, la generación de las mismas de una forma automática constituye una de las tareas más importantes de esta Tesis. Por ello, varios algoritmos serán explicados a lo largo del capítulo 4 para automatizar la aplicación del Control Distribuido a las rutas de datos.



(a) Unidad de Generación de Enable del controlador de M1



(b) Generación de las señales de rutado y carga del registro R1

Figura 9: Arquitectura canónica de las UGEs y de las señales de rutado y carga de registros aplicada al benchmark DiffEq

0.4.3. Ejemplo de aplicación del Control Distribuido

La mejor manera de comprender los conceptos previamente mencionados es aplicarlos. En esta sección se verá cómo aplicar la arquitectura del Control Distribuido al benchmark DiffEq, así como un ejemplo de su flujo de ejecución. Para ello se partirá de la planificación y asignación mostradas anteriormente en la figura 1.

0.4.3.1. Ejemplo de arquitectura del Control Distribuido

La figura 9 contiene la implementación de la UGE asociada al controlador del multiplicador M1, así como la generación de las señales de rutado y carga del registro R1.

En primer lugar consideremos la implementación de e_{M1}, mostrada en la figura 9a. Recordemos que e_{M1} se encarga de verificar el Teorema de

State	$routing_{R1}$
St_{M1}=S1	0
St_{M2}=S3	1
St_{M1}=S7	0

Tabla 3: Tabla de verdad para generar la señal de control del multiplexor asociado al registro R1

Ejecución Especulativa para las operaciones asignadas a M1. Así pues, e_{M1} es la **AND** de 2 señales: hit_{M1} (condición 1) y la salida de un multiplexor controlado por St_{M1} (condición 2). Las entradas de este multiplexor son las líneas de resolución de dependencias de las *Operaciones 1*, *6* y *7* (condición 3). Consideremos por ejemplo la *Operación 6*, que tiene riesgos de tipo RAW y WAW con la *Operación 2*, ejecutada en el multiplicador M2. Dichos riesgos serán resueltos si St_{M2}=*S3*, ya que implicará que la *Operación 2* ha finalizado. Nótese que también existe una dependencia RAW con la *Operación 1*, pero para evaluar las dependencias con las operaciones asignadas a la misma UFE bastará con saber el valor de la propia variable de estado del controlador, en este caso St_{M1}.

A continuación veamos cómo implementar las señales de control del registro R1 (véase la figura 9b). Primero nótese que las *Operaciones 1*, *3* y *7* están asignadas a R1. Para identificar plenamente que una operación puede ser escrita, es necesario utilizar el enable de la UFE donde ha sido ejecutada, y la variable de estado del controlador de dicha UFE. Por ello, la señal de carga $load_{R1}$ es la **OR** de tres multiplexores, cada uno de ellos controlado por la variable de estado donde la operación en concreto ha sido ejecutada, y cuyas entradas son los enables de las UFEs. Por ejemplo, cuando la *Operación 3* sea escrita St_{M2} valdrá S3 y e_{M2} será '1'.

Para seleccionar qué UFE escribirá en R1 en cada momento, será preciso construir una tabla de verdad como la mostrada en la tabla 3. M1 y M2 son las UFEs que escriben en R1. Sin pérdida de la generalidad, supongamos que M1 utilizará la entrada 0 del multiplexor y M2 la entrada 1. Así pues, la señal de control valdrá '1' solo cuando St_{M2}=S3. Por tanto utilizaremos la condición St_{M2}=S3 para implementar $routing_{R1}$.

0.4.3.2. Ejemplo de ejecución del Control Distribuido

La figura 10 muestra la ejecución de 2 iteraciones del benchmark DiffEq, utilizando UFEs monociclo y el Control Distribuido. Por ello, en la parte derecha de la figura hay varias columnas, ya que cada UFE tendrá su propio controlador. En concreto, hay 2 columnas por UFE. La primera es la de la variable de estado, y la segunda columna es la que se corresponde con el biestable-T asociado a cada controlador, que aportará información relativa

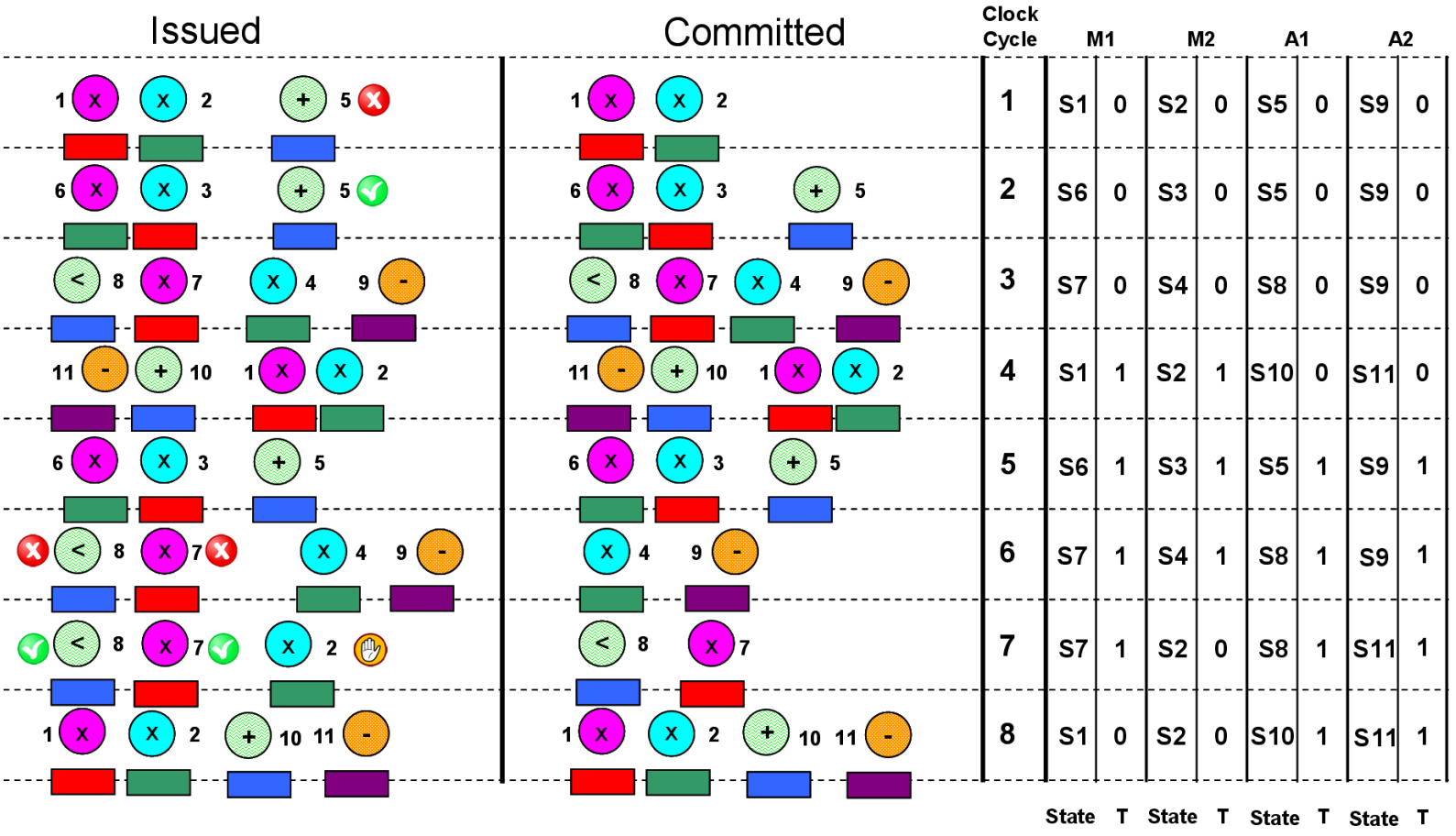


Figura 10: Ejecución del benchmark DiffEq con UFEs monociclo y Control Distribuido

a la iteración, y cuyo uso será visto en profundidad en el capítulo 4 de esta Tesis. De hecho, para los propósitos de este apartado, se obviarán los cambios en dicho biestable. Nótese que se supondrán los mismos errores de predicción que en el ejemplo de la figura 6, en la que se utilizaba el Control Centralizado.

En el ciclo 1 hay un fallo en la *Operación 5*, asignada al sumador A1. Como es independiente de las *Operaciones 1* y *2*, dichas operaciones se escriben en sus respectivos registros, mientras que el controlador del sumador A1 se queda parado en estado S5 un ciclo más. Por ello, en el ciclo 2 se corrige la *Operación 5*, planificada estáticamente en el cstep 1, mientras que se ejecutan las *Operaciones 6* y *3*, planificadas estáticamente en el cstep 2.

Continuando con el flujo de ejecución, se llega a que los errores en las *Operaciones 8* y *7*, estáticamente planificadas en los csteps 2 y 3, respectivamente, suceden ambos en el ciclo 6. Como no afectan a la finalización de las *Operaciones 4* y *9*, solamente se paran los controladores de A1 y M1. En el ciclo 7, las *Operaciones 8* y *7* son corregidas, mientras que la *Operación 2* tiene que esperar por una dependencia de tipo WAR con la *Operación 10*, que ni tan siquiera ha sido lanzada en A1.

Finalmente, 2 iteraciones del benchmark DiffEq han sido completadas en 8 ciclos. Si suponemos que el tiempo de ciclo del caso especulativo es 0.75 veces el del no especulativo, como en el Control Centralizado, tendremos un tiempo de ejecución de 8 * 0.75 = 6 unidades de tiempo. Es decir, es un 25 % más rápido que el caso convencional. Nótese que el rendimiento en realidad es algo mayor, ya que además de las 2 iteraciones, las *Operaciones 1* y *2* de la tercera iteración también se han completado.

En esta sección se ha mostrado la arquitectura básica y funcionamiento del Control Distribuido. A lo largo de la Tesis la arquitectura se verá con mayor profundidad, evaluándose con detalle los problemas que aparezcan, y se introducirán pequeñas modificaciones que permitirán la utilización de UFEs multiciclo y encadenamiento.

0.4.4. Resultados experimentales

La eficiencia del Control Distribuido será demostrada experimentalmente en esta sección. El tiempo de ejecución y el área serán comparados con implementaciones convencionales basadas en Ripple Carry Adders y Carry Select Adders y con una implementación con UFEs y Control Centralizado.

Se ha construido un simulador para medir la latencia media de los circuitos, ya que en el caso de las UFEs depende de los valores reales. El parámetro p ha sido utilizado para medir la correlación de los datos, probándose con tres valores: 0.5 (ninguna correlación), 0.75, 1 (máxima correlación). Los circuitos han sido sintetizados con la herramienta comercial *Synopsys Design Compiler*, con una librería de 65 nm, obteniéndose los tiempos de

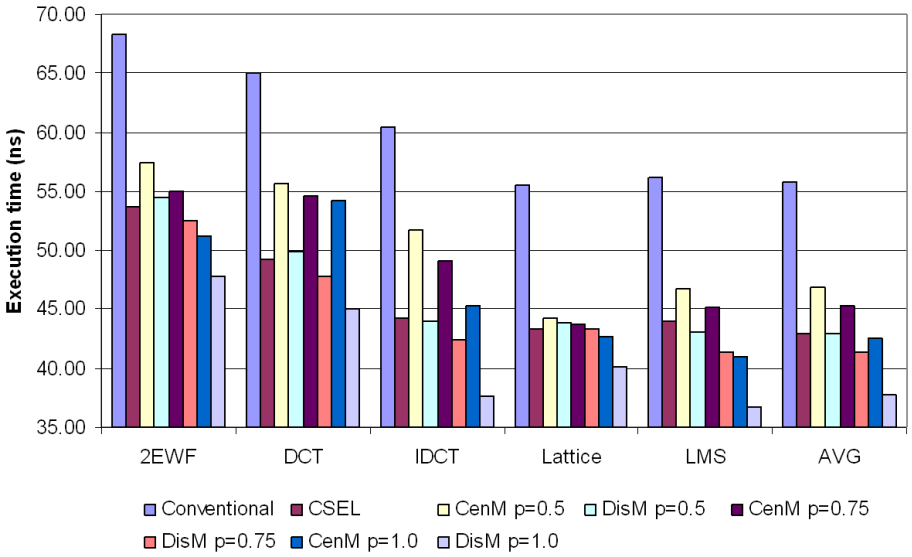


Figura 11: Tiempo de ejecución con UFEs multiciclo y encadenamiento de operaciones

ciclo y resultados de área. Se han probado 6 benchmarks, con una precisión de 16 bits:

(1) *Differential Equation* (DiffEq).

(2) *Second Order Elliptic Wave Filter* (2EWF).

(3) *Discrete Cosine Transform* (DCT).

(4) *Inverse Discrete Cosine Transform* (IDCT).

(5) *Lattice Filter* (Lattice).

(6) *Least Mean Square Filter* (LMS).

Nótese que en las figuras relativas al tiempo de ejecución, los resultados del benchmark DiffEq se han omitido para mejorar la visualización de todos los resultados, al permitir un mayor nivel de detalle.

El apéndice A de la Tesis contiene más información sobre el marco de trabajo utilizado en los experimentos.

La figura 11 muestra los tiempos de ejecución de las diversas implementaciones. En este experimento se han considerado UFEs multiciclo y encadenamiento de operaciones. Tal como puede observarse, la implementación con Control Centralizado (CenM) puede reducir el tiempo de ejecución en un

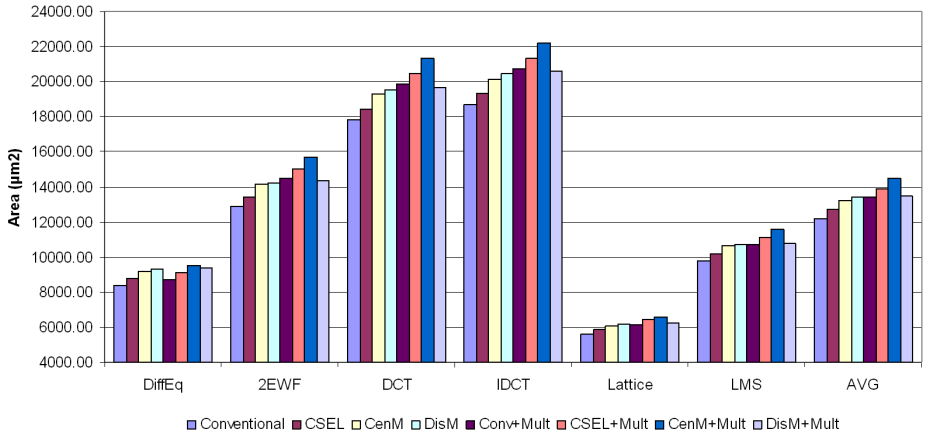


Figura 12: Área con diferentes estilos de implementación

24.2 % en promedio, con respecto a la implementación convencional, obteniendo resultados similares a los de la implementación con CSELs. Sin embargo esto solo sucede cuando la correlación es muy alta, i.e. p=1. Por ejemplo, con p=0.75 esta mejora solo es del 18.7 %, y un 5.3 % peor que la implementación con CSELs.

La implementación con Control Distribuido (DisM) mejora los resultados del CenM. En el caso peor, i.e. p=0.5, esta mejora es del 23 % y del 8.4 % con respecto a una implementación convencional y otra con CenM. Además, el rendimiento es aproximadamente el mismo que el de la implementación con CSELs. En el caso mejor, i.e. p=1.0, las mejoras aumentan hasta el 32.5 %, 11.5 % y 12.5 % con respecto a las implementaciones convencional, CenM y CSEL, respectivamente.

Además, nótese que estos resultados han sido obtenidos con técnicas convencionales de planificación y asignación de recursos. En la sección 4.5 de esta Tesis se presentarán técnicas de Síntesis de Alto Nivel capaces de explotar mejor las características del Control Distribuido.

La figura 12 contiene los resultados de área de los circuitos. Se han comparado 4 estilos de implementación: el convencional basado en RCAs, el convencional basado en CSELs, el Control Centralizado y el Control Distribuido. Cada uno de estos estilos ha sido probado a su vez con UFs monociclo y multiciclo.

Los resultados muestran que en promedio las implementaciones CSEL, CenM y DisM ocupan un 4 %, 8.8 % y 10.2 %, respectivamente, más que la implementación convencional, en el caso de las UFs monociclo. Pero en el caso multiciclo esta penalización es del 3.6 %, 7.8 % y del 0.6 % respectivamente. El motivo es que utilizar UFEs multiciclo con el Control Distribuido tiene

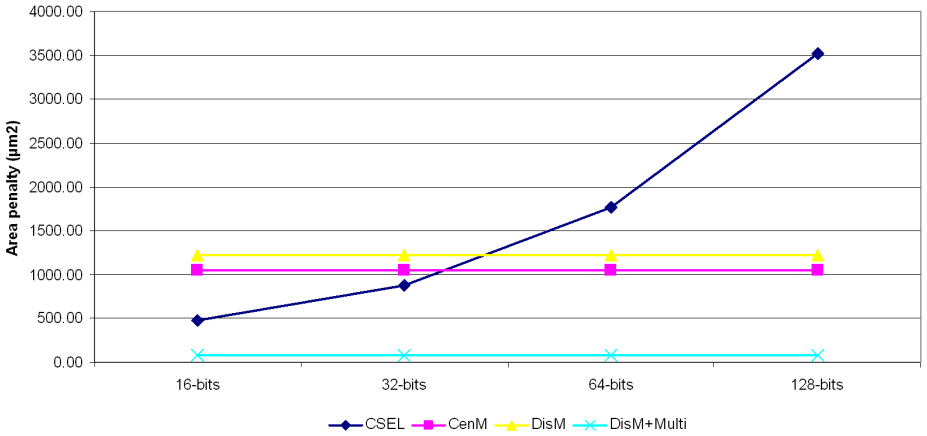


Figura 13: Penalización del área con respecto a la anchura de datos

una penalización mucho más baja que en el caso convencional o con Control Centralizado, en los que el controlador global aumenta mucho el número de estados.

Además de estos resultados de área, debe tenerse en cuenta que la penalización debida a los Controles Centralizado y Distribuido se debe a los predictores y al propio control, mientras que en el caso de los Carry Select Adders se debe a la anchura de datos. Este hecho queda reflejado en la figura 13. Esta figura muestra la evolución de la penalización con respecto a la anchura de datos. Se muestran 4 implementaciones: con CSELs, con Control Centralizado, y con Control Distribuido mono y multiciclo. En los casos CSEL y CenM solo se considera la implementación monociclo, ya que las penalizaciones monociclo y multiciclo son parecidas. Tal y como puede observarse, para tamaños grandes la penalización por el uso de CSELs es mucho mayor que en el caso de CenM y DisM. Nótese que la penalización en el caso del Control Distribuido con UFEs multiciclo es más pequeña que con UFEs monociclo porque lo que se muestra es la diferencia de área con respecto al caso base multiciclo y monociclo, respectivamente.

0.5. Conclusiones y trabajo futuro

Históricamente todo ha sido estático en Síntesis de Alto Nivel. Para optimizar los ASICs lo máximo posible se aplican técnicas de pre-síntesis y/o síntesis. Todo puede controlarse estáticamente porque se imponen ciertas restricciones a la hora de aplicar dichas técnicas. Estas restricciones por tanto, pueden limitar las posibilidades de la SAN. Una de ellas consiste en asumir que la latencia de las UFs es fija.

Sin embargo, con la aparición de las Unidades Funcionales de Latencia Variable, se rompe esta asunción y se abre un abanico de posibilidades. Las UFLVs pueden finalizar una operación en diferentes números de ciclos. Este es el caso de las Unidades Funcionales Especulativas, que son una implementación concreta de UFLVs cuya latencia depende de adivinar el valor de una o varias señales internas. En esta Tesis se proponen dos diseños de UFEs. Las UFEs combinan un buen rendimiento con una pequeña penalización por área. Por este motivo pueden ser extremadamente útiles en SAN.

Sin embargo, las UFEs no tienen sentido si no pueden usarse. Las UFEs trabajan más rápido si aciertan en la predicción. Por tanto, este hecho debe ser tenido en cuenta por el controlador de la ruta de datos. Los trabajos previos, como las *Unidades Telescópicas* presentadas en [BMPM98], proponen modificar directamente el controlador replicando estados en aquéllos casos donde se utilicen estas UFLVs. Esto incrementará exponencialmente la complejidad del controlador, por lo que solamente muy pocas UFLVs podrán ser utilizadas.

En esta Tesis se proponen dos técnicas para afrontar este problema. Primero el Control Centralizado, propuesto en el capítulo 3. Consiste en detener completamente la ruta de datos cada vez que haya un error en la predicción, en cualquiera de las UFEs. El principal problema es que a medida que se incrementa el número de UFEs, la probabilidad de parada será mayor. De cara a mitigar este problema y dejar que la ruta de datos continúe su curso, se propone el Control Distribuido, en el capítulo 4. La idea es detener tan solo las operaciones que sufren el fallo en la predicción, y aquéllas que dependen de éstas, permitiendo que el resto continúe su ejecución.

La penalización de área causada por estas alternativas es bastante pequeña. Además, el área empleada por la implementación con Control Distribuido es prácticamente idéntica considerando UFEs monociclo o multiciclo. Este hecho compensa la penalización por área debido al control de las UFEs, con respecto a implementaciones convencionales, la cual es menor que en el caso de utilizar módulos más complejos como los Carry Select Adders.

En esta Tesis se definen arquitecturas genéricas para ambas técnicas y se presenta la base formal para certificar el buen funcionamiento de los circuitos generados. Se ha desarrollado un entorno de trabajo que permite la simulación de los circuitos con valores reales, ya que el rendimiento de las UFEs depende de dichos valores, no puede ser medido directamente con la planificación y tiempo de ciclo dados por las herramientas comerciales. Además, se proponen diversos algoritmos que permiten la inclusión de los Controles Centralizado y Distribuido en el marco del Diseño Automático.

Los resultados experimentales confirman que con estas técnicas de gestión es posible integrar un gran número de UFEs en las rutas de datos, consiguiendo un rendimiento mejor que el de las implementaciones no especulativas, especialmente si se utiliza el Control Distribuido.

En conclusión, en esta Tesis se ha desarrollado un método de síntesis para integrar las Unidades Funcionales Especulativas en el flujo de la Síntesis de Alto Nivel. Ahora que la *ingeniería* ha sido desarrollada, la *arquitectura* puede incorporar diferentes *ladrillos*. En otras palabras, ahora que se ha establecido la metodología, nuevas Unidades Funcionales Especulativas pueden ser incorporadas automáticamente.

En el futuro, deben construirse nuevas ideas sobre la base presentada en esta Tesis, y de hecho están en la mente del autor y algunas ya son trabajo en desarrollo:

(1) La construcción de un modelo de potencia para comprobar la eficiencia energética de las técnicas propuestas en esta Tesis.

(2) El diseño de Unidades Funcionales Multiespeculativas. Si con un predictor es posible reducir el tiempo de ejecución, el uso de muchos predictores lo disminuirá mucho más?

(3) Asignación dinámica. Si es posible planificar dinámicamente, por qué no asignar dinámicamente?

(4) Unidades Funcionales Multiciclo con Latencia Dinámicamente Variable. Tendrá sentido ? Será posible modificar dinámicamente la latencia de las Unidades Funcionales?

Nótese que todas estas técnicas de carácter dinámico solamente son aplicables y tienen sentido si se parte de una planificación dinámica, al menos desde el punto de vista del rendimiento. Por ejemplo, independientemente de la asignación, una ruta de datos tendrá el mismo rendimiento si está usando el Paradigma de Ejecución No Especulativo, ya que el número de csteps está determinado por la planifiación (estática). De este modo, la asignación dinámica carece de sentido en un contexto no especulativo. O por ejemplo, modificar la latencia de una UF solo será posible en un contexto de planificación dinámica, ya que cada cambio en la latencia de la UF afectará a la planificación inicial. Por tanto, y para concluir, el lector debería valorar las oportunidades que el trabajo presentado en esta Tesis ofrece de cara al futuro.

0.5.1. Publicaciones

A continuación, he aquí una lista con las publicaciones producidas durante estos años de investigación:

(1) A.A. Del Barrio, S. Ogrenci Memik, M.C. Molina, J.M. Mendías and R. Hermida. "Power Optimization in Heterogeneous Datapaths". In Proc. *Design, Automation and Test in Europe (DATE)*. 2011, pp. 1400-1405.

(2) A.A. Del Barrio, S. Ogrenci Memik, M.C. Molina, J.M. Mendías and R. Hermida. "A Distributed Controller for Managing Speculative Functional Units in High Level Synthesis". In Proc. *IEEE Trans. On Computer-Aided Design of Integrated Circuits and Systems (TCAD)*. 2011, pp. 350-363.

(3) A.A. Del Barrio, S. Ogrenci Memik, M.C. Molina, J.M. Mendías and R. Hermida. "Using Speculative Functional Units in High Level Synthesis". In Proc. *Design, Automation and Test in Europe (DATE)*. 2010, pp. 1779-1784.

(4) M.C. Molina, R. Ruiz-Sautua, A.A. Del Barrio and J.M. Mendias. "Subword Switching Activity Minimization to Optimize Dynamic Power Consumption". *IEEE Design & Test of Computers*. 2009, pp. 68-77.

(5) A.A. Del Barrio, M.C. Molina, J.M. Mendías, E. Andrés, G. Botella, R. Hermida and F. Tirado. "Applying branch prediction techniques to implement adders". In Proc. *Conference on Design of Circuits and Integrated Systems (DCIS)*. 2008.

(6) A.A. Del Barrio, M.C. Molina, J.M. Mendías, E. Andrés, R. Hermida and F. Tirado. "Applying speculation techniques to implement functional units". In Proc. *IEEE International Conference on Computer Design (ICCD)*. 2008, pp. 74-80.

(7) A.A. Del Barrio, M.C. Molina, J.M. Mendías, E. Andrés and R. Hermida. "Restricted chaining and fragmentation techniques in power aware high level synthesis". In Proc. *Euromicro Conference on Digital System Design (DSD)*. 2008, pp. 267-273.

(8) E. Andrés, M.C. Molina, G. Botella, A.A. Del Barrio and J.M. Mendías. "Area Optimization of Combined Integer and Floating Point Circuits in High-Level Synthesis". In Proc. *Southern Programmable Logic Conference (SPL)*. 2008.

(9) E. Andrés, M.C. Molina, G. Botella, A.A. Del Barrio and J.M. Mendías. "Aerodynamics Analysis Acceleration Through Reconfigurable Hardware". In Proc. *Southern Programmable Logic Conference (SPL)*. 2008.

(10) A.A. Del Barrio, M.C. Molina, J.M. Mendías, and R. Hermida. "Pattern-guided switching minimization in high level synthesis". In Proc. *Conference on Design of Circuits and Integrated Systems (DCIS)*. 2007, pp. 175-180.

(11) A.A. Del Barrio, M.C. Molina and J.M. Mendías. "Bit-level Power Optimization during Behavioural Synthesis". In Proc. *Conference on Design of Circuits and Integrated Systems (DCIS)*. 2006.

(12) A.A. Del Barrio, M.C. Molina and J.M. Mendías. "Optimización a nivel de bit del consumo de potencia durante la Sintesis de Alto Nivel". In Proc. *Jornadas de Paralelismo*. 2006, pp. 621-626.

Chapter 1

Introduction

The man with a new idea is a Crank
until the idea succeeds.

Mark Twain, in "Following the Equator"

Eratosthenes of Cyrene was a Greek mathematician, astronomer, geographer, poet, music theorist and even an athlete. He invented the armillary sphere, proposed his popular sieve for finding prime numbers and contributed to the development of science and technology in his time. But today he is known as the first human being able to calculate the Earth radius and, hence, circumference. It was 240 BC, there were no machines. But human intellect has always gone beyond any difficulty. He abstracted the problem and with the aid of several estimations and the trigonometric properties, Eratosthenes calculated the Earth circumference with a 1 % error.

Along Human History there have been many people that have contributed to evolve society. Like Eratosthenes they abstracted the problems of their respective times and created new solutions. The first modern car was invented in the XIX$^{\text{th}}$ century, the first computer in the early XX$^{\text{th}}$ century, and nowadays in the XXI$^{\text{th}}$ century life is no longer conceived without a laptop or a mobile phone.

Current technology is dominated by these devices, which constitute the main challenge of our time. Albert Einstein said once *"Only two things are infinite, the universe and human stupidity, and I'm not sure about the former"*. We are not sure about the two of them, but modestly I believe he forgot to mention one more candidate as infinite thing: human ambition.

Users want more powerful electronic devices, with longer batteries and duration and smaller sizes. The *Complementary Metal Oxide Semiconductors* (CMOS) appearance has helped to comply with all these constraints. The continuous decrease of the channel width in the target technology has made it possible to reduce execution time, area and power, in spite of some coun-

terparts, as increasing power density for example. However this solution *only* proposes to diminish the size of technology. Nowadays the CMOS channel width is reaching its lower bounds, and going down 45 nm is really difficult in the *Application Specific Integrated Circuits* (ASICs) industry because it is very expensive and because of the increase on leakage, its exponential effect on temperature and the consequent lose of reliability.

Nevertheless human beings can comply with users' necessities from higher abstraction levels. These more abstract solutions have the ability of being applied over a wider range of cases, since higher-level ideas do not depend on the lower levels, while lower-level solutions make the higher-level design decisions be dependant on them. For instance, imagine an architect trying to design a 100-floored building. He can think in the huge building and then choose steel for constructing the structure. However, if he thinks about using wood or bricks, he will only be able to build a cabin or a small block of flats.

On the other hand, high-level abstractions sometimes lack of important information to achieve the best solutions. For example, imagine the architect trying to design the same building in Japan or San Francisco. The building must be earthquakes resistant, so the architect should think about elastic materials to dissipate the earthquake force.

In the ASICs context, engineers do not have to design buildings, fortunately, but they must search for the best circuit to comply with users' necessities. This Ph.D. Thesis will try to satisfy these requirements from the High-Level Synthesis point of view, and like the architect, the proposed solution will take into account some special features of the *bricks* that will implement the initial specification given by the designer. In concrete, conventional implementations will be accelerated while keeping a low area overhead thanks to the adaptation of the High-Level Synthesis flow to the special features of a new kind of Variable Latency Functional Units.

1.1. High-Level Synthesis

High-Level Synthesis (HLS), sometimes referred to as C synthesis, *Electronic System Level* (ESL) synthesis, algorithmic synthesis, architectural-level or behavioral synthesis, is an automated design process that interprets an algorithmic description of a desired behavior and produces a hardware description that implements that behavior [CM08] while complying with some constraints given by the designer.

Although authors may classify differently and tools may execute in different order, HLS is divided in four main tasks:

(1) *Partitioning*, that divides a behavioral description into sub-descriptions in order to diminish the size of the problem or to satisfy external constraints.

(2) *Allocation*, which is the task of assigning operations onto available functional unit types (available in the target technology).

(3) *Scheduling*, which assigns operations to control steps in order to satisfy the designer constraints.

(4) *Binding*, which assigns operations to specified instances of unit types.

However, as partitioning produces sub-specifications that must be allocated, scheduled and bound, we can only consider points 2, 3 and 4 as the authentic HLS tasks. Besides this task division, one of the most controversial points in HLS has always been the election of the specification language. A language must gather several conditions for specifying properly the circuits and making easier and more efficient the information processing. This will be examined deeper in subsection 1.1.1.

HLS has been a major preoccupation of *Computer-Aided Design* (CAD) researchers since the late 1970s. Early work in HLS examined scheduling heuristics for dataflow designs. The most straightforward approaches include scheduling all operations *As Soon As Possible* (ASAP) and scheduling the operations *As Late As Possible* (ALAP) [KT85, TS86, Mar86, Tri87]. These were followed by some heuristics that used metrics such as urgency [Gir84] and mobility [PK89] to schedule operations. Other ideas consisted in rescheduling the designs iteratively [PK91]. Research in resource allocation and binding techniques have targeted different goals like reducing registers, the used *Functional Units* (FUs), wire delays and interconnect costs [Mic94, CW91, KT85]. Later on, considering the interdependent relationship that exists among these tasks within HLS, researchers focused on realizing them in parallel, basically through approaches using *Integer Linear Programming* (ILP) [HLH91, GE92, LMD94, WMGB95]. Afterwards, taking into account the direct impact of how control structures affected the quality of the synthesized circuits, authors increased their efforts on handling more complex control flows [HJH+93].

New techniques have appeared during the last years, as the use of compiler transformations, which can further improve HLS, although they were originally developed for improving code efficiency for sequential program execution. For instance, this is the case of *Common Subexpression Elimination* (CSE) and copy propagation, which are commonly seen in software compilers [KR07]. However, although these basic transformations can be used in synthesis, other transformations need to be adapted in order to incorporate ideas of mutual exclusiveness of operations, resource sharing, and hardware cost models. Later attempts in the early 2000's tried to overcome limitations on concurrency inherent in the input algorithmic descriptions [Lab03].

Nowadays, as technology progresses and systems become more and more complex, the use of high-level abstractions is increasing its importance in the design task. On the one hand designers' performance augments, because most

of the optimizations can be carried out automatically by CAD tools, reducing thus design time. On the other hand, HLS provides to hardware something similar to what the JAVA language provides to software, i.e. portability of solutions. Independently of the target platform, a scheduling, binding, etc. that reduces latency and the needed resources is always a good solution.

However solutions given by HLS algorithms must be examined carefully. One algorithm aiming at the minimization of cycle time, can increase the overall area at an unaffordable price, or the opposite effect, performance can be degraded while trying to minimize power consumption, as this is quite influenced by the frequency, i.e. the cycle time inverse. All these considerations will be evaluated in the subsection 1.1.2.

1.1.1. Specification of the problem

In 1974 [BS74, BS73], Mario Barbacci noticed that in theory one could *compile* the instructions set of a processor, then using the *Instruction Set Processor Specification* (ISPS) language, [BSG+77, Bar81] into hardware. This was the first notion of design synthesis from a high-level language specification. In later years this design synthesis concept evolved and by the early 1980s, the fundamental tasks, of what would be named HLS, had been decomposed into hardware modelling, scheduling, resource allocation, binding and control generation. However this decomposition was performed for solving specific problems, although almost all of these subtasks are interdependent. It is not until 1988 when McFarland, Parker and Camposano presents their *"Tutorial on High-Level Synthesis"* [MPC88] at the Design and Automation Conference. They defined HLS as the transformation from a behavioral specification to a *Register Transfer Level* (RTL) structure that realizes the given behavior, and established a serie of basic interrelated tasks, namely: *scheduling*, *allocation* and *binding*.

First generation behavioral synthesis tools was introduced by Synopsys in 1994 as Behavioral Compiler [Ber95], and used Verilog or VHDL as input languages. Tools based on *Hardware Description Languages* (HDLs) were not widely adopted because they were not suited to model behavior at high level. In fact, ten years later Synopsys discontinued the developement of Behavioral Compiler.

In 2004 a new generation of commercial HLS tools emerged, and provided then synthesis of circuits specified at C level to a RTL specification. Synthesizing from the popular C language offers abstraction, expressive power and coding flexibility while tying with existing flows and models. Moreover HLS has added some new features to C-like languages such as the use of bit-widths in the input specifications. This language shift, combined with other technical advances has been a key for its successful industrial usage.

Nevertheless, independently of the input language in order to convert a

high level language into hardware, a *Control Dataflow Graph* (CDFG) is a fundamental element to be used. The CDFG owns a natural parallelism that helps to optimize the final circuit. It is basically a graph with nodes (i.e. vertices) and edges. A CDFG is the result of combining two graphs: the *Dataflow Graph* (DFG), including operations and data dependencies, and the *Control-flow Graph* (CFG), including conditional branching, iteration, and module.

E.A. Snow was the first to present the use of CDFGs in HLS in his Ph.D. Thesis [Sno78]. After him and during the 1980s and the early 1990s, several CDFG approaches appeared in [OG86, CR89, CT89, DeJ91]. Finally, from 1990s on, most of the HLS works and tools adopted CDFGs as the intermediate representation structure [PK89, ZG99, WWB+02, CS02, MM05, KMO+08].

Nevertheless, the initial behavioral specification is given in a higher-level language, such as C or VHDL. Hence, several tools have been developed aiming at the extraction of the CDFGs from this initial high-level language. CDFG Toolkit [JAC02], developed by the Seoul National University, can support fast and easy generation and manipulation of CDFGs, mainly for HLS. CDFG Toolkit is generally used as an input format to a HLS system. It includes a CDFG generator, a CDFG to C/VHDL converter, a CDFG parser, and a CDFG viewer. CHESS [NRV03], developed by Namballa et al., is a tool for CDFG extraction and HLS of *Very Large Scale Integration* (VLSI) systems. It starts from an initial VHDL specification and performs several compiler-level transformations, followed by a series of behavioral-preserving transformations, and finally the CDFG is extracted. The ChipCflow project [LM09] is a system where a C program is initially converted into a CDFG, and then to VHDL. After generating the complete VHDL program, an *Electronic Design Automation* (EDA) tool converts it into a bitstream which is downloaded in a *Field Programmable Gate Array* (FPGA).

This Ph.D. Thesis will be focused on data intensive applications, so the DFG structure will be used in order to represent the initial functional specifications. In this way, the proposed techniques will be applied independently from the initial language specification. Hence, in order to complete a HLS framework, the previous use of a tool like CDFG will be enough for generating the DFG and then use the system proposed in this Ph.D. Thesis.

1.1.2. Performance and cost

Human ambition becomes infinitum, as it was explained in the very beginning of this chapter. A user cannot imagine that his laptop is going to play a movie slower because it is saving power; a designer cannot take out several FUs from the circuit because the area constraints are violated, if with this removal the applications cannot process information as fast as it was expected. Hence, although there are many problems where diminishing

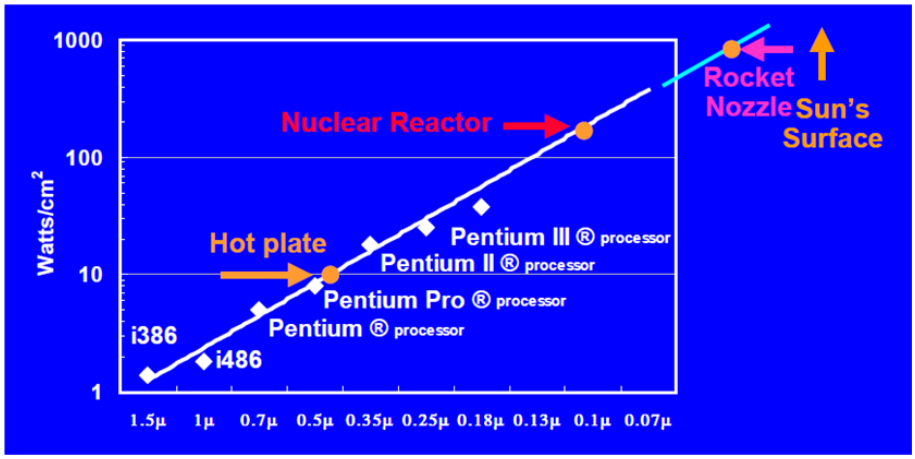


Figure 1.1: Moore's Law, technology size vs power density. Source: S. Borkar, R. Ronen, F. Pollack, IEEE Micro-1999

area or power, increasing reliability, etc. constitutes the target goal, they must respect the initial performance expectation.

This is the reason why historically Computer Science has been focused on increasing performance, until some years ago. Moore's Law [Moo65] has predicted Computer Science behavior until these days. For example it states, among several other assumptions, that frequency and the number of transistors in processors are doubled every two years. Thereby, performance has been scaled quite well, because it is proportional to frequency, as it can be seen in equation 1.2. However, power density has been increased exponentially too, as we get closer to the sub-micron technologies. This is depicted in figure 1.1. As power density augments, chips degrade faster. Therefore, improving performance is still important, but not at any cost.

$$T_{execution} = CPI \times T_{cycle} = \frac{CPI}{f} \tag{1.1}$$

$$Performance = \frac{1}{T_{execution}} = \frac{f}{CPI} \tag{1.2}$$

On the other hand this power dissipation is dependant on the logic and processing resources on the chip. This is because of a huge variation in the silicon efficiency. A large number of studies have revealed that energy or area efficiency for a given function realized on a silicon substrate can vary by two to three orders of magnitude. For example, the power efficiency of a microprocessor-based design is typically 100 million operations per watt, where as reprogrammable arrays (such as FPGAs) can be 10-20x, and a custom ASIC can give another 10x gain or more. In a recent study, Kuon and Rose show that ASICs are 35x more area efficients than FPGAs [KR07].

If done right, *Integrated Circuit* (IC) design offers the possibility of 10-100x gain in silicon efficiency. However, efficiency does not always mean performance. Efficiency is *the capability for achieving a concrete target.* For example consider the wide range of different implementations of a single module, e.g. an adder. Literature offers single designs like *Ripple Carry Adders* (RCAs) or more complex designs such as *Carry Lookahead Adders* (CLAs) [Kor02]. RCAs are small and low-power, but slow, while CLAs are faster, but very large and power-hungry. Therefore if area is the target function, RCAs will be more efficient than CLAs. Or for instance, into the purest HLS context, think about a scheduling that takes 3 or 4 less *control-steps* (csteps) but it introduces a huge FU in the design. This solution will be efficient or not depending on the performance and area constraints. In other words, can the user accept this increase in terms of area? Does he really want a faster mp3 player? Will the user buy a faster device if he cannot carry it in his pocket? Or maybe what the user wants is a faster device with a similar or even a smaller area?

Therefore the techniques used during the IC design must often satisfy more than one constraint at the same time. Hence, improving the quality of a parameter must not degrade very much the rest of them. In this way, the use of *balanced* components can help to comply with a set of *balanced* constraints. For instance, *Speculative Functional Units* (SFUs) are datapath elements that combine a good performance with a low area/power overhead. This feature can be used in two senses:

(1) On the one hand they can achieve a better performance with a low area overhead, with respect to the proposed solution.

(2) or on the other hand they can obtain the desired performance with a lower area overhead than with other existing solutions.

Hence, SFUs are good candidates to be included in the HLS flow in order to comply with a balanced set of constraints. However it is not trivial how to integrate SFUs in HLS. SFUs can execute operations with a different delay, depending on the inputs. They can be included in the modules library to be used in the allocation process as any other FU, but their behavior varies dynamically. So they do not fit to the static features of the HLS. The answer to this question and the appearance of new problems will be evaluated and solved during the following chapters of this Ph.D. Thesis.

1.2. Non Speculative Execution Paradigm

Before going deeper into the ideas of speculation, the main topic of this Ph.D. Thesis, in this subsection the reader is invited to think about what

cstep	M1	M2	A1	A2	R1	R2	R3	R4
1	1	2	5		1	2	5	
2	6	3	8		3	6	8	
3	7	4		9	7	4		9
4			10	11			10	11

Table 1.1: DiffEq binding summary

conditions are necessary for writing operations in the registers of a datapath without speculation.

HLS is composed by a set of static techniques that generate both a datapath and a controller. The DFG is scheduled, allocated and bound and its operations are executed in the cstep given by the controller state, in the FU where they were bound. Every state transition is defined statically, so everything happens as it was scheduled, i.e. an operation scheduled in cstep C, is always executed in cstep C. And this happens independently of the target FUs.

Common FUs such as RCAs, *Carry Select Adders* (CSELs), CLAs, etc. [Kor02] have a static behavior. Their execution time is always the same independently of the inputs, so they fit perfectly to the traditional HLS techniques. Some years ago *Variable Latency Functional Units* (VLFUs) appeared in order to increase performance at low cost [BMM+08c, WDH01, ADH05, Cil09, VBI08, BMPM98, RRL00]. Some works as [WDH01, ADH05, VBI08, Cil09, BMM+08c] only propose FUs designs. Nonetheless, the real interesting thing is not the design, but to use it. However in all the previous attempts to incorporate VLFUs to the HLS flow, such as [BMPM98, RRL00], every possible behavior is taken into account in the datapath controller. Hence, there always exists an state transition which decides what operations must be executed in every cstep.

The objective of this subsection is to illustrate what conditions are necessary to perform this state transition, that is always defined statically, in other words, what conditions are necessary to write an operation result in its corresponding register. Note that this question is not often thought over, because conventional HLS techniques guarantee that these conditions will be satisfied. However, the comprehension of them will be very important to utterly understand the principles of the speculative paradigm, defined later in chapter 4.

In order to illustrate these conditions, for example let's consider the *Differential Equation* (DiffEq) DFG, which is shown in figure 1.2, with a possible scheduling and binding. In order to make the figure easier to understand, the binding summary is depicted in table 1.1. There are two adders(A1-A2), two multipliers (M1-M2), and four registers (R1 through R4). Each opera-

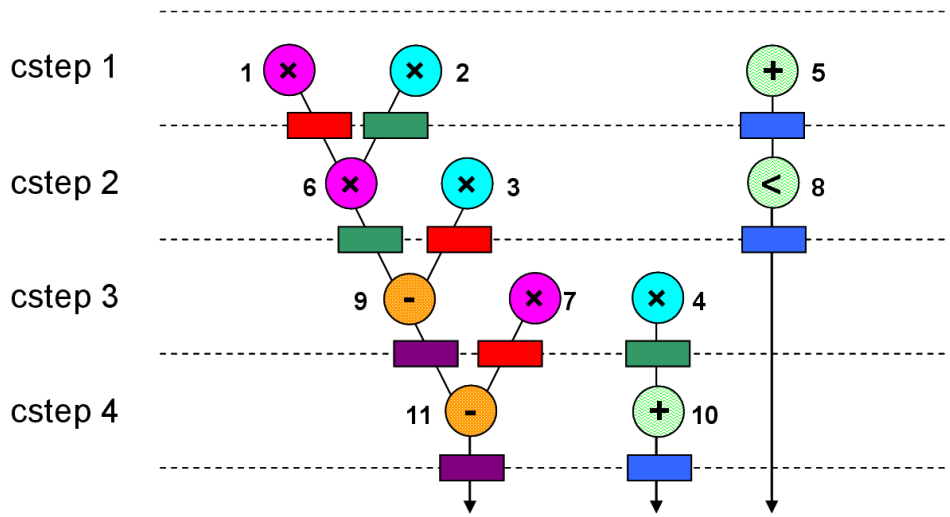


Figure 1.2: DiffEq scheduling and FU and register binding

tion is identified by a number (1 through 11). Let's consider *Operation 6*. The question is ... when can *Operation 6* be written in register R2? *Operation 6* will be written in R2 if the datapath controller is in state 2 and if its predecessors, i.e. *Operations 1* and *2*, have been executed and written before. Actually, the state condition is enough, because to be in state 2 implies that *Operations 1* and *2* have been committed before, but for the practical purpose of this subsection, which is no other that claryfing later the Speculative Execution Paradigm, the division between the state condition and the dependencies condition will be maintained. A theorem will be developed in chapter 3 in order to capture all these conditions.

1.3. HLS and Speculation

Traditional datapath implementations cannot work with SFUs. Their fundamental execution paradigm is completely static, which is not suitable for handling mispredictions in SFUs. In a datapath containing SFUs, mispredictions must be corrected in execution time to certify correct results and to keep the datapath in the correct state. Therefore this must be taken into account when performing the HLS.

Correcting mispredictions without adversely impacting overall performance is the most important challenge. The performance of a SFU is dependant on the predictor hit rate. The hit rate can be high for an individual SFU. However, if the datapath contains several SFUs, the probability of predicting all the carries correctly decreases rapidly. Speculative adders and multipliers considered in this Ph.D. Thesis will use only one predictor

per unit. Therefore, supposing a hit rate p for every single predictor, the probability of n modules correctly guessing all the carries will be p^n. Consider a datapath with four adders and two multipliers, and suppose that the probability of guessing a carry is 90 % per single predictor. The global probability of guessing all the carries is $(0.90)^6$, i.e., 53 %. Although data correlation will increase the hit-probability, frequent occurrence of mispredictions will diminish overall performance. Hence, the datapath management scheme should aim to maximize performance.

Two management techniques will be presented in this Ph.D. Thesis. On the one hand the Centralized Management, which stops the whole datapath everytime there is a failure in whatever the FU, and on the other hand the Distributed Management, that will only stop those FUs involved in the failure.

1.3.1. Related Work

SFUs have a longer latency when there is a misprediction. The main idea is to operate them with a low misprediction rate, such that their lower latency thanks to speculation can be exploited in the average case. This is similar to VLFUs. VLFUs will be referred as the more general term for various functional unit designs, which can exhibit different latencies for the same operation. Some VLFUs are optimized to operate faster on certain data values. Others make use of various prediction schemes to accelerate part of the computation path. SFUs belong to this later category.

In the approach explained in [Mue99], an age-based scheduler working with VLFUs at instruction-level granularity was proposed. However this granularity is only applicable to processors, and the use of FIFOs results in large area and power overhead from the point of view of HLS, where the datapaths must be highly optimized for complying with the constraints given by the designer.

On the other hand, new synthesis techniques have been developed to integrate VLFUs into datapaths. A static rescheduling technique is proposed for using variable latency modules [RRL00]. The DFG is transformed into a CDFG, which causes a significant increase in terms of area and only allows the use of a limited number of VLFUs. Besides, the VLFUs utilized in this work are significantly larger than their fixed latency counterparts. *Telescopic Units* [BMPM98] introduced a paradigm to automatically build variable-latency circuits. An error detection function, referred as *hold function*, is computed to inform the system at which cycle the correct result will be available at the outputs.

Another approach is to generate variable latency circuits by reducing the critical path with *speculative points* searching in the netlist [BCK09]. A speculative point is a node in the netlist where speculation can diminish the

delay. However, none of the previous approaches addresses the problem of using many VLFUs in the same circuit and in different points of the static scheduling.

The use of several VLFUs produce many different possible schedulings that must be considered statically, according to the conventional HLS techniques [Mic94, CM08]. For instance, think about a circuit with S SFUs which are used in the same cstep. It is possible that none of them produces a misprediction, or it can happen that only SFU_1 mispredicts, or it can happen that SFU_1 and SFU_2 fails, etc. Every combination of hits and failures will produce a different scheduling. In general, there will be 2^S possibilities that will have to be controlled in that cstep, depending on which SFUs are working in low-latency or high-latency modes, i.e. depending on inputs that happen in execution time, dynamically. Therefore, the number of possible cases that can happen when using VLFUs (and concretely SFUs for the purposes of this Ph.D. Thesis), is really huge to be controlled with conventional techniques.

1.3.2. Basic ideas and a motivational example

In order to illustrate the advantages of adapting the architecture to the use of SFUs and the necessity of a new execution paradigm with this kind of units, consider the DiffEq example with monocycle FUs. Two iterations of the DiffEq algorithm will be run. The scheduled DFG has been shown in figure 1.2. The binding to the adders (A1-A2), multipliers (M1-M2) and registers (R1-R4) has been depicted in table 1.1. In every execution example four columns will be shown: on the leftmost part the *Issued* and *Committed* operations are depicted, i.e. those operations that have solved their RAW and structural-FU hazards and those that can be written in their corresponding registers, respectively, and on the rightmost part both the clock cycle and controller state are shown.

Firstly, consider the execution flow of a baseline implementation without any speculative FU. See figure 1.3. Since there is no prediction, the state of execution is always the same as the cstep dictated by the static scheduling. In this example, 8 cycles are required to complete 2 iterations, so the total execution time can be derived as $T_{ex} = 8*1 = 8$ *time units*.

Next the use of the two techniques investigated in this Ph.D. Thesis will be illustrated. Figures 1.4 and 1.5 depict the cycle-by-cycle execution flow for the same example for two consecutive iterations using SFUs. In figure 1.4 and 1.5, Centralized and Distributed Management schemes are used, respectively. Monocycle SFUs have been supposed. Therefore if there is a hit in the prediction, SFUs will take one cycle, while if there is not, they will take two. Mispredictions are indicated by a ✘ symbol, corrections by a ✔ symbol, and a hand denotes that the operation stalls in that cycle.

The cycle time when using SFUs will be assumed to be 75 % of the

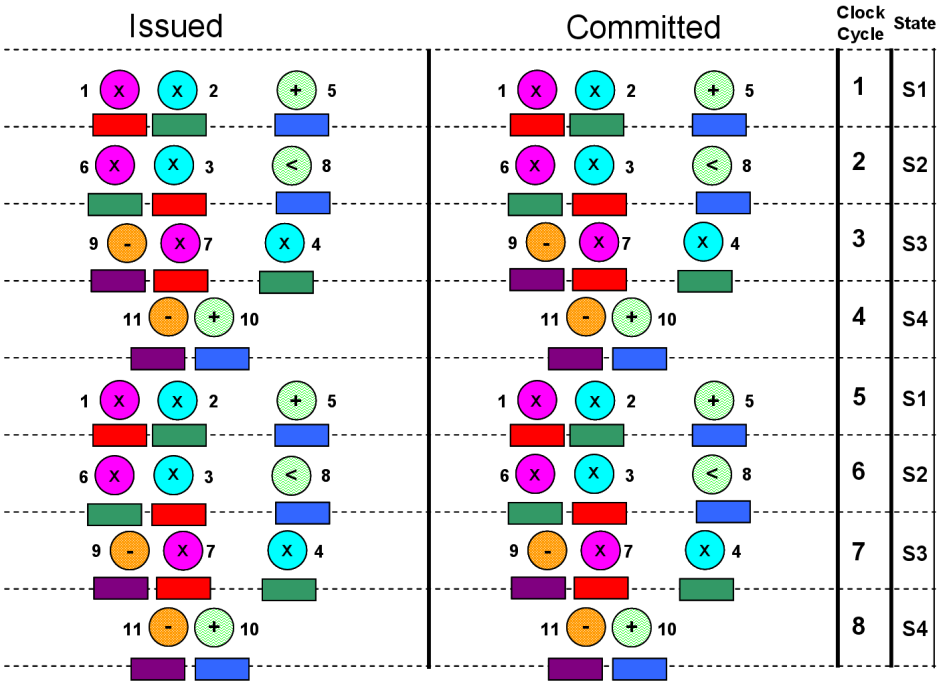


Figure 1.3: Issued, committed operations and controller evolution in the DiffEq example with common implementation

cycle time without using speculative units. The carry path of the speculative multiplier described in [BMM+08c] is around 75 % of a common multiplier carry path. Since in homogeneous datapaths multiplier delay is likely to be dominant, the cycle time should be proportional to it.

Now, let's consider figure 1.4. Centralized Management scheme is assumed here, which stops the whole datapath every time there is a SFU misprediction. There are three mispredictions in this illustrative example, so there will be three penalty cycles. Note how all operations scheduled in a cstep where there is a misprediction are not allowed to write. The execution time in this case will be T_{ex} = 11*0.75 = 8.25 *time units*. This is worse than the baseline case. The reason is that stopping the whole datapath introduces too many penalty cycles, such that the reduced latency of SFUs cannot compensate for them.

Finally, see figure 1.5. Distributed Management is used here, which only stops the datapath partially for those computations affected by the misprediction, while the rest of the datapath continues its execution. Note, for instance, how the misprediction in *Operation 5* does not affect *Operations 1* and *2*, that can be written into registers R1 and R2, respectively. In the same way, the correction of *Operation 5* does not affect *Operations 3* and *6*.

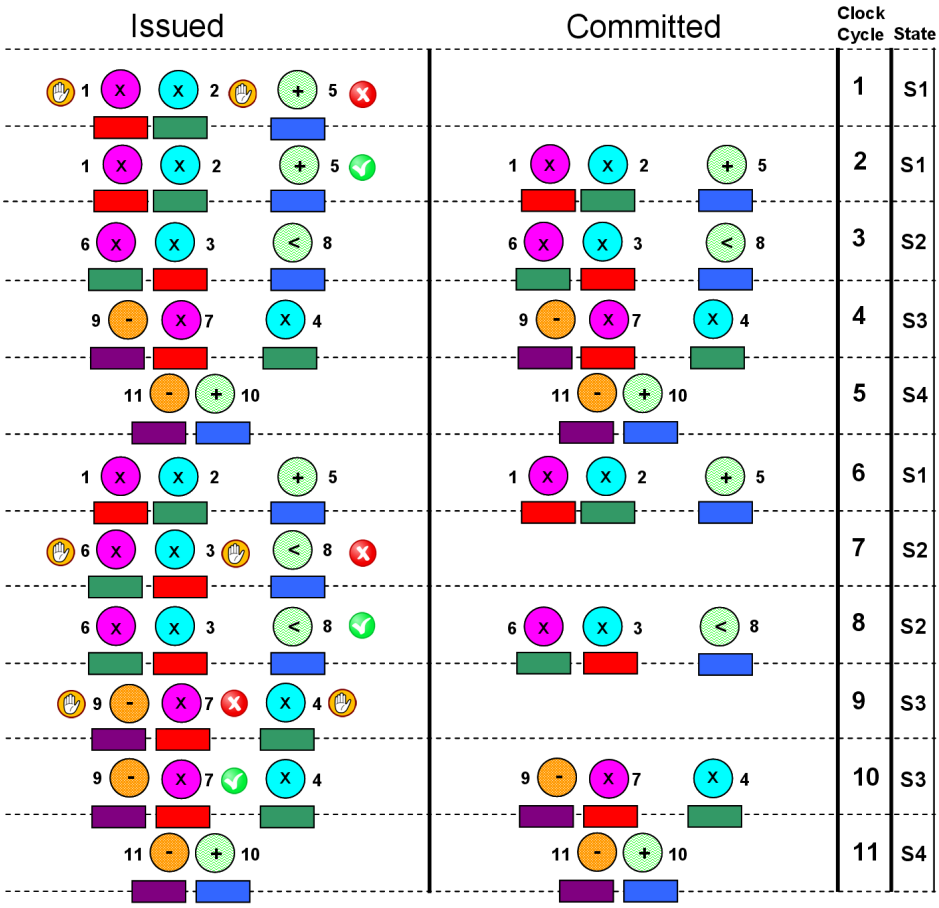


Figure 1.4: Issued, committed operations and controller evolution in the DiffEq example with SFUs and Centralized Management

Therefore, in cycle 2, two operations statically scheduled in cstep 2 and one in cstep 1 are being executed simultaneously. The same mispredictions as in the Centralized Management case occur with *Operations 7* and *8* in the second iteration, which are both corrected in cycle 7, while only *Operation 2* (actually belonging to iteration 3) remains stalled. Overall, a total of two iterations and 2 more operations (*Operations 1* and *2*) of the third iteration, will be committed in a total execution time of T_{ex} = 8*0.75 = 6 *time units*, which is better than the non-speculative baseline case.

The example presented above illustrates the potential benefits of using SFUs. At the same time, it indicates the need for a careful management scheme for the datapaths. In the following chapters of this Ph.D. Thesis, the proposed control algorithms for dynamically managing SFUs will be presented and developed in depth. Both Centralized and Distributed Management

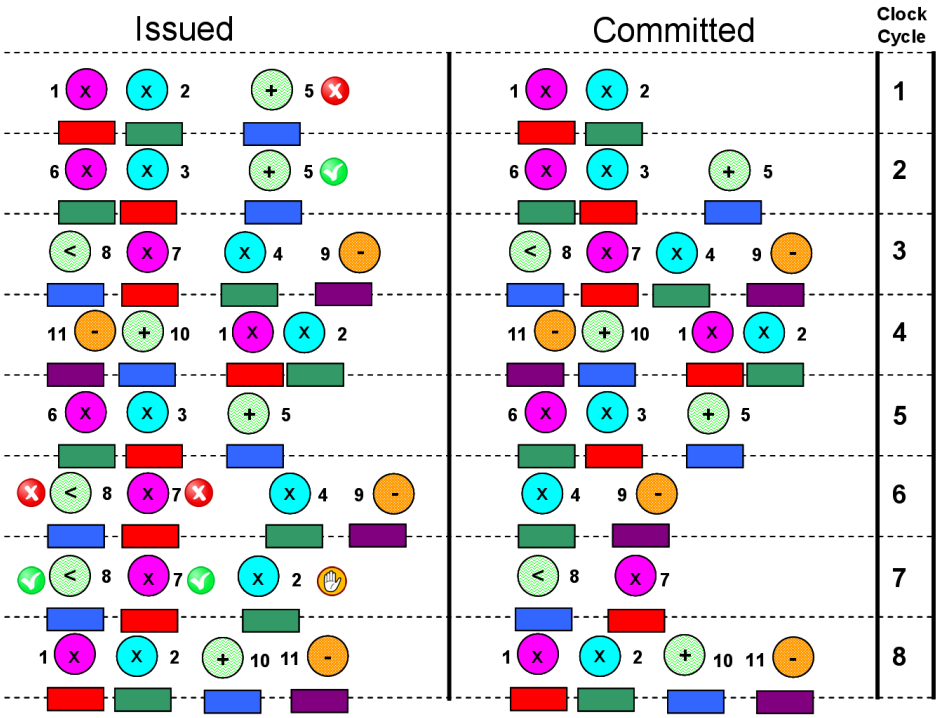


Figure 1.5: Issued, committed operations and controller evolution in the DiffEq example with SFUs and Distributed Management

will take advantage of the SFUs special features in order to minimize the total execution time.

1.4. Objectives of this Ph.D. Thesis

The automation of design process has become necessary because of the increasing complexity of designs and the decreasing time to market requirements. In this automation process, the modules that have been traditionally used for implementing circuits always work with a fixed latency, so all the HLS flow is based on this assumption. The appearance of VLFUs suppose a change in this point of view. However, all the previous works have consisted in either designing a pretty fast FU which works many times in the low-latency case [WDH01, ADH05, Cil09, VBI08], or in using a very little number of VLFUs for not increasing excessively the controller complexity [BMPM98, RRL00].

Speculative Functional Units are a subset of VLFUs and consist in arithmetic functional units that operate using some prediction logic in order to diminish the latency of the module. In concrete, the SFUs that will be pre-

sented in this Ph.D. Thesis try to predict the carry signal to shorten the critical path of the functional unit. The average case performance of these units is determined by the hit rate of the prediction. In case of mispredictions, the SFUs need to be coordinated by some mechanism in order to perform corrections and to maintain the datapath in a correct state.

The main reason for using this kind of functional units is the great tradeoff between performance and area/power penalty that they offer. Traditional HLS approaches try to satisfy the critical path constraint imposed by the designer with faster modules like CLAs or CSELs. However their area/power overheads can easily produce a violation of these constraints. SFUs reduce critical path but without increasing area too much. For instance, designs described in this Ph.D. Thesis achieve a CSEL-like performance while keeping a RCA-like area. However the main purpose of this Ph.D. Thesis will not be the SFUs design, but their control instead. As the best feature of SFUs is the performance vs area/power tradeoff, the additional logic for controlling every possible misprediction must be complex enough for keeping datapath in a correct state, but simple enough to compensate the use of SFUs instead of larger common functional units with smaller latency.

Therefore, devising a control mechanism for correcting mispredictions without adversely impacting overall performance, and without incurring a high area/power penalty, is the most important challenge. In this Ph.D. Thesis the necessary techniques for generating a datapath controller for the deployment of SFUs in HLS are presented. Moreover, on the one hand several theorems and lemmas that prove the correctness of the used techniques are formulated and demonstrated, and on the other hand the concrete architecture that will implement them is depicted. In addition, after integrating SFUs in the HLS flow, some synthesis techniques will be presented in order to take advantage of the special features of these implementations, and improve thus the performance of the final circuits with a negligible or null area overhead.

1.5. Thesis outline

The rest of the thesis is organized as follows: chapter 2 reviews the traditional FU designs and introduces some SFU designs. Chapter 3 presents the foundations of the Centralized Management: a first naïve management for handling the mispredictions. Chapter 4 describes the Distributed Management: a smarter management for improving performance in comparison to the one presented in the previous chapter. Besides it presents the formal demonstration of the theorems and lemmas in order to justify the correctness of this technique. In addition to this, it explains some HLS optimizations that can be performed in order to take advantage of the special features of the Distributed Management. Our concluding remarks will be given in chapter

5, as well as the future lines of work. Finally, the appendixes A and B will explain and show some details that are more related with the implementation task.

Chapter 2

Functional Units design

A journey of a thousand miles begins
with a single step

Chinese Proverb

Designs that include arithmetic units have proliferated in recent years. In this chapter the most basic FU designs are briefly described. All the algorithms and designs that have ever been suggested are not, and could not be, included. Of course this escapes from the scope of this Ph.D. Thesis. Besides these algorithms are meant to serve as a solid introduction to the Computer Arithmetic rich field, which is continously evolving. The knowledge of these structures will help the reader to better understand the speculative FUs [BMM+08a, BMM+08c] that are described at the end of sections 2.1.6 and 2.2.3 and that will be used later in this Ph.D. Thesis.

These speculative FUs are based on the prediction of internal signals, by using predicting structures similar to those utilized in processors when predicting branches. Delay and area results will be shown and discussed at the end of this chapter.

Note that only parallel adders and multipliers will be explained. On the one hand, every more complex operation can be reduced to a set of additions and products. On the other hand, parallel designs have been chosen because most of the techniques used in HLS are developed considering this kind of modules. Moreover, pipelined FUs are similar to combinational ones, but introducing some latches for dividing the execution in several stages.

Finally, as it has been said in the introduction, in spite of describing several FUs in this chapter, the reader must be clear about the target of this Ph.D. Thesis, which is not the FUs design, but their later use in the HLS flow.

2.1. Adders design

Addition is the key arithmetic operation in most of the digital circuits and processors. Therefore, their performance and other parameters, such as area and consumption, are highly dependent on the adders features. Multipliers and other complex modules usually include a great amount of adders. Although the memory accesses are the main bottleneck in actual processors [WM95], the increase in adders performance becomes critical in the design of ASICs.

Historically there have been several proposals to implement modules capable to execute additions. In this section the most basic techniques to implement adders will be described, ranging from the simple Ripple Carry Adders to the most complex designs, such as the Carry Lookahead Adders [Kor02]. Finally some speculative adders will be described in order to tackle the problem of achieving a good performance while keeping a low area penalty. Some of these modules will be used in later chapters and in the multipliers design section.

2.1.1. Ripple Carry Adders

The most straightforward implementation of a parallel adder for two operands X=$x_{n-1}x_{n-2}\ldots x_0$ and Y=$y_{n-1}y_{n-2}\ldots y_0$ is through the replication of n basic units called *Full Adders* (FAs). A Full Adder is a logical circuit with three inputs, namely: two operand bits, say x_i and y_i, and an incoming carry bit, denoted by c_i; and two outputs: the corresponding sum bit, denoted by s_i and an outgoing carry bit, denoted by c_{i+1}. The outgoing carry c_{i+1} becomes the incoming carry for the following FA, the one that has x_{i+1} and y_{i+1} as input bits. The FA is a combinational circuit that implements the following Boolean equations:

$$s_i = x_i \oplus y_i \oplus c_i \tag{2.1}$$

where $\oplus$ is the logic **XOR** operation, and

$$c_{i+1} = x_i \cdot y_i + c_i \cdot (x_i + y_i) \tag{2.2}$$

where $x_i \cdot y_i$ is the logic **AND** operation and $x_i + y_i$ is the logic **OR** operation.

In a parallel arithmetic unit, all the *2n* input bits (X and Y) are usually available to the adder at the same time. However, the carries have to propagate from position 0, corresponding to the input bits x_0 and y_0, to position i in order to generate s_i and c_{i+1}. Therefore, it is necessary to wait until the carries *ripple* through the whole adder, i.e. the n FAs. Because of

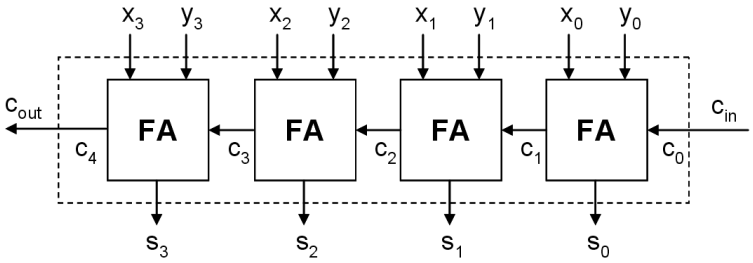


Figure 2.1: 4-bits Ripple Carry Adder

this reason, this adder structure is known as *Ripple Carry Adder* (RCA). In other words, the execution time of a RCA is $\mathbf{O}(\boldsymbol{n})$.

Note that unless chained adders are being used, in every additive operation the carry-in to the whole adder, c_0 is always '0', so the FA corresponding to this position can be simplified. The resulting circuit is called *Half Adder* (HA) and its Boolean equations can be derived by substituting c_i by '0' in equations 2.1 and 2.2.

To summarize this subsection, it can be concluded that a RCA is a simple and regular adder. It is easy to design and possesses a low area and power overhead. However it has the counterpart of the execution time, which is too slow in general.

2.1.2. Carry Select Adders

Carry Select Adders (CSELs) implement the most basic strategy in order to improve the performance of RCAs. In a CSEL the n bits are divided into non-overlapping groups of possibly different lengths. Each group generates two sets of sum bits and an outgoing carry. A set assumes that the incoming carry into the group is '0', while the other assumes that it is '1'. Supposing that there are m groups, $m < n$, when the outcoming carry of the previous group is generated, say group *k-1*, $k \leq m$, it selects the set of sum bits and carry-out of the k^{th} group. Therefore a multiplexer per group is needed for selecting the sum bits and carry-out that have been calculated with the correct carry-in. Finally, note that every submodule, which adds every group of bits, can be implemented with whatever technique. Although in the case of CSELs, the most common combination is to divide a big adder into several submodules implemented with the RCA technique and interconnected with the CSEL technique.

A couple of CSEL implementations are depicted in figure 2.2 and 2.3. Figure 2.2 shows a uniform [Bed62] 8-bits CSEL, which is composed of two 4-bits adders. Note how the most significant submodule is replicated in order to perform the addition with both carries-in '0' and '1'. Besides a multiplex-

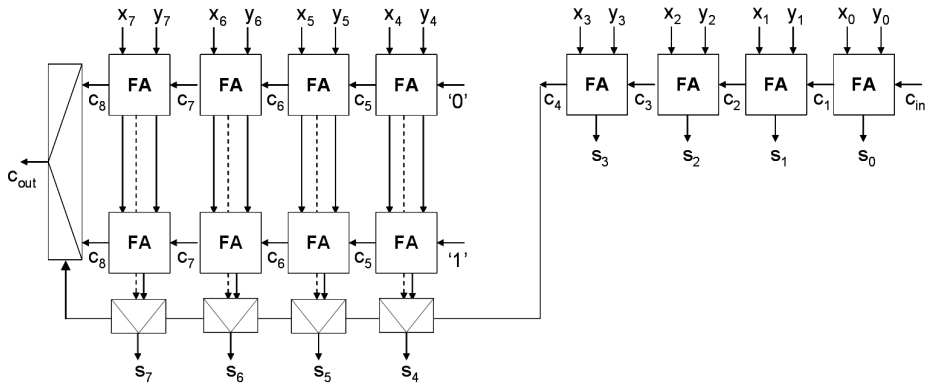


Figure 2.2: Uniform 8-bits Carry Select

er and certain logic are included for selecting properly the sum bits and the carry-out when the carry-in from the least significant module, i.e. c_4, is available. 16-bits CSELs similar to the structure shown in 2.2, but composed of two 8-bits RCAs instead, will be used in the experiments presented in chapters 3 and 4.

Figure 2.3 shows an implementation of a variable [Tya93, ARI10] 11-bits CSEL, that is, the size of every group is different. In this case every sub-module consists of a RCA. The foundation of variable CSELs is to increase performance. In order to build a variable CSEL, the size of the k^{th} group must be chosen so as to match the delay within the group and the delay of the carry-select chain from group *0* to group *k-1*. With this strategy the group lengths follow the simple arithmetic progression 1, 2, 3, ... This happens for every group except for group *0*, which has length 1. Therefore, supposing m groups, the total number of bits, n, must satisfy

$$\begin{aligned} 1 + (1 + 2 + 3 + \ldots + (m-1)) &\geq n \\ 1 + m(m-1)/2 &\geq n \\ m(m-1) &\geq 2n - 1 \end{aligned} \tag{2.3}$$

As a result of equation 2.3, the size of the largest group and the execution time of the CSEL are $\mathbf{O}(\sqrt{n})$. For example, with $n = 32$, based on equation 2.3, nine groups are required. A possible choice for their sizes is 1, 1, 2, 3, 4, 5, 6, 7 and 3. Although this timing analysis has been performed for variable CSELs, it is valid for uniform CSELs too.

Compared to the RCA, the CSEL requires duplicated carry-chain logic and additional carry-select logic, which supposes a great area and power overhead, in spite of the improvement in terms of performance.

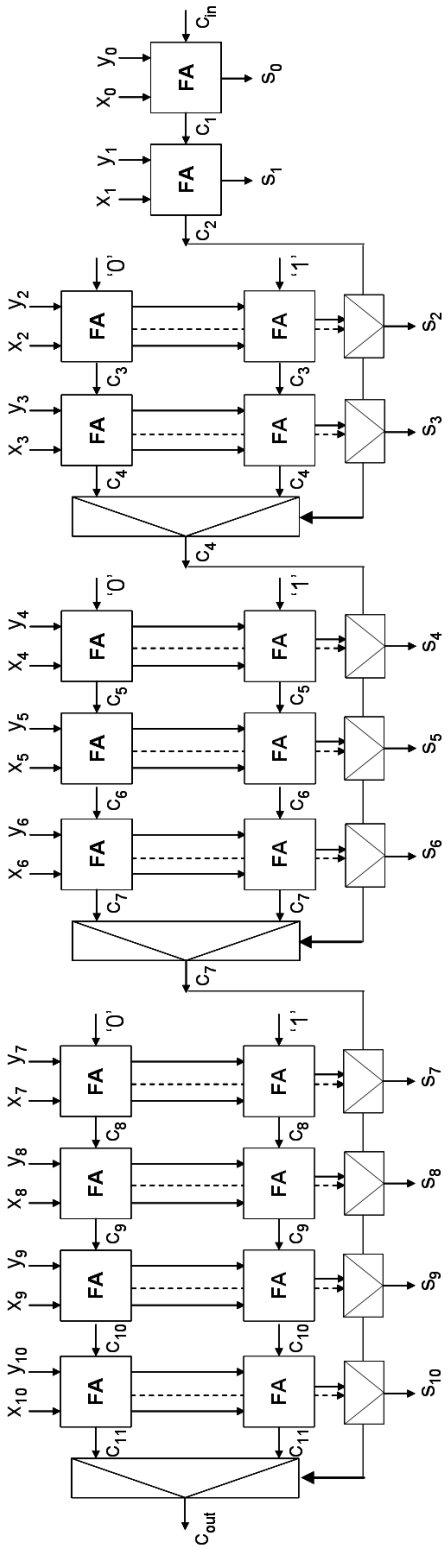


Figure 2.3: Variable 11-bits Carry Select

Conditional Adders [Skl60] and Carry-Skip Adders [Kor02, ARI10] can be considered as variations of the CSELs, but they will not be treated deeply because the FU design is not the target of this this Ph.D. Thesis. Conditional Adders try to apply the CSEL principle recursively, i.e. divide the n bits into two groups of *n/2* and interconnect them with CSEL technique, then divide each group of *n/2* into two groups of *n/4* and interconnect them with CSEL technique, and so on. In this way, is possible to achieve an execution time $\mathbf{O(\log(n))}$, although the area overhead is greater than with the traditional CSELs. Carry-Skip Adders divide the bits in several groups, uniform or variable such as the CSELs, but they do not replicate submodules, using instead additional logic for calculating if every submodule is propagating the carry or not. Execution time is similar to traditional CSELs, i.e. $\mathbf{O(\sqrt{n})}$. More CSEL ideas can be found in hybrid structures such as [DWA+92, LJ92].

2.1.3. Carry Lookahead Adders

The carry lookahead technique is the most commonly used scheme in order to accelerate the carry propagation, which determines adders performance. The main idea behind *Carry Lookahead Adder* (CLA) is to generate all incoming carries in parallel (for all the *n-1* high order FAs) and avoid the need to wait until the correct carry propagates from the FA where it has been generated. This objetive is achievable, since it must be taken into account that on the one hand carries depend on the input bits $X{=}x_{n-1}x_{n-2}\ldots x_0$ and $Y{=}y_{n-1}y_{n-2}\ldots y_0$, and on the other hand both X and Y are available to all stages of the adder. However, building a truth table to evaluate all possible cases and therefore generate $c_n \ldots c_1$ requires too much logic and constitutes an impractical approach.

The carry lookahead technique try to reduce the amount of logic needed for knowing each carry-in, by deciding for every stage if this will generate and/or propagate a carry. The i^{th} stage will generate a carry-out equal to '1' if both input bits, say x_i and y_i, are equal to '1', independently of the carry-in c_i. In the same way, if both input bits are equal to '0', the resulting carry-out will be '0'. On the other hand, if x_i and y_i are different, the resulting carry-out c_{i+1} will be '0' if c_i is '0' and '1' if c_i is '1'. Hence when x_i and y_i are different, the i^{th} stage propagates c_i to c_{i+1}. This behavior can be summarized with the Boolean expressions shown in equations 2.4 and 2.5

$$g_i = x_i \cdot y_i \tag{2.4}$$

$$p_i = x_i \oplus y_i \tag{2.5}$$

However, the **XOR** gate depicted in equation 2.5 can be reduced to a single **OR** gate, as shown by equation 2.6. When both bits are '1', the i^{th}

stage will generate a carry, i.e. g_i='1', so it will not matter if p_i ='1'.

$$p_i = x_i + y_i \tag{2.6}$$

Combining equations 2.4 and 2.6 with equation 2.2, c_{i+1} can be expressed now as

$$c_{i+1} = x_i \cdot y_i + c_i \cdot (x_i + y_i) = g_i + c_i \cdot p_i \tag{2.7}$$

Taking into account equation 2.7, it can be asserted that $c_i = g_{i-1} + c_{i-1}p_{i-1}$. If this is substituted in equation 2.7, then equation 2.8 is obtained

$$c_{i+1} = g_i + g_{i-1}p_i + c_{i-1}p_{i-1}p_i \tag{2.8}$$

Further substitutions results in equation 2.9

$$\begin{aligned} c_{i+1} &= g_i + g_{i-1}p_i + g_{i-2}p_{i-1}p_i + c_{i-2}p_{i-2}p_{i-1}p_i = \dots \\ &= g_i + g_{i-1}p_i + g_{i-2}p_{i-1}p_i + \dots + c_0p_0p_1 \dots p_i \end{aligned} \tag{2.9}$$

The expression given by equation 2.9 allows us to calculate all the carries in parallel. However, as it has been mentioned at the beginning of this subsection, using this equation for all the carries results impractical. Therefore, designs in literature, like the ones presented in [Kor02, ARI10], combine carry lookahead technique with the ripple carry one, mainly. For example, the *Ripple-Block Carry Lookahead Adder* (RCLA) uses several modules, implemented as CLAs, that are interconnected with the ripple carry technique. On the other hand, the *Block-Carry Lookahead Adders* (BCLA) uses RCA blocks interconnected with the carry lookahead technique. Note that in both cases the typical size of the submodules is 4-bits.

The next step in the CLAs evolution is to apply several levels of carry lookahead, in order to accelerate carry propagation with a tree-like structure. For example, a 16-bits CLA can be implemented with two levels of lookahead, using 4-bits CLA submodules, whose carry lookahead expressions are shown in equation 2.10, and a second carry lookahead level, whose target is to generate the input carries to those submodules.

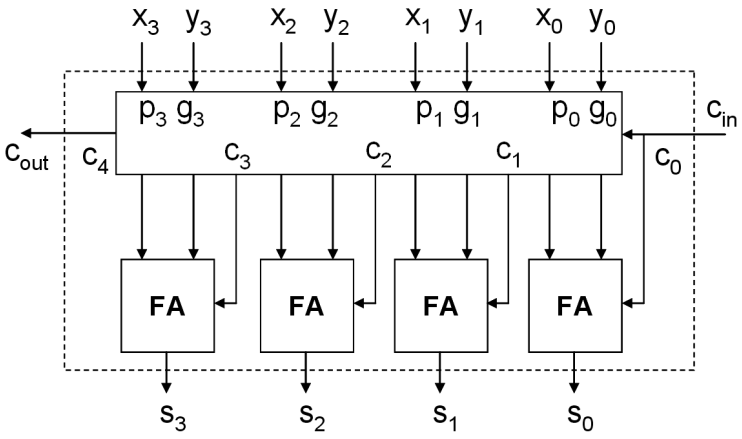
$$\begin{aligned} c_1 &= g_0 + c_0p_0 \\ c_2 &= g_1 + g_0p_1 + c_0p_0p_1 \\ c_3 &= g_2 + g_1p_2 + g_0p_1p_2 + c_0p_0p_1p_2 \\ c_4 &= g_3 + g_2p_3 + g_1p_2p_3 + g_0p_1p_2p_3 + c_0p_0p_1p_2p_3 \end{aligned} \tag{2.10}$$

In order to generate the c_{4i} carry-in's, generate and propagate group signals are required. Using 4-bits submodules, the group signals are given by

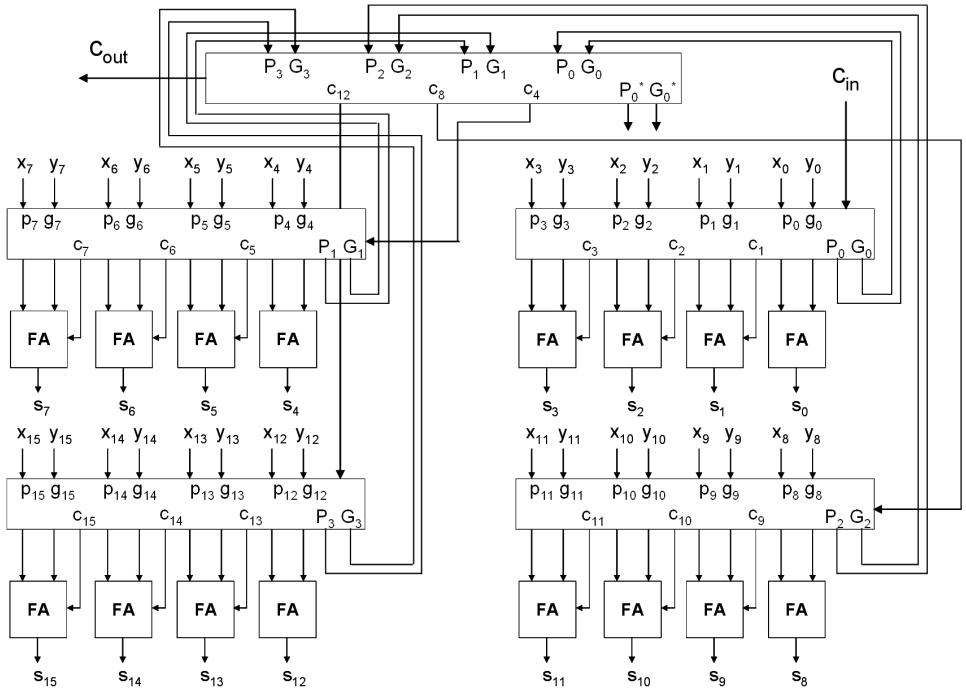
equation 2.11

$$\begin{aligned} G &= g_3 + g_2p_3 + g_1p_2p_3 + g_0p_1p_2p_3 \\ P &= p_0p_1p_2p_3 \end{aligned} \tag{2.11}$$

Having these group signals, equation 2.10 can be applied for obtaning



(a) 4 bits 1-level Carry Lookahead Adder



(b) 16 bits 2-levels Carry Lookahead Adder

Figure 2.4: Carry Lookahead implementations

c_{4i}'s. Note how in equation 2.12 the group signals are used instead of each internal generate/propagate signal.

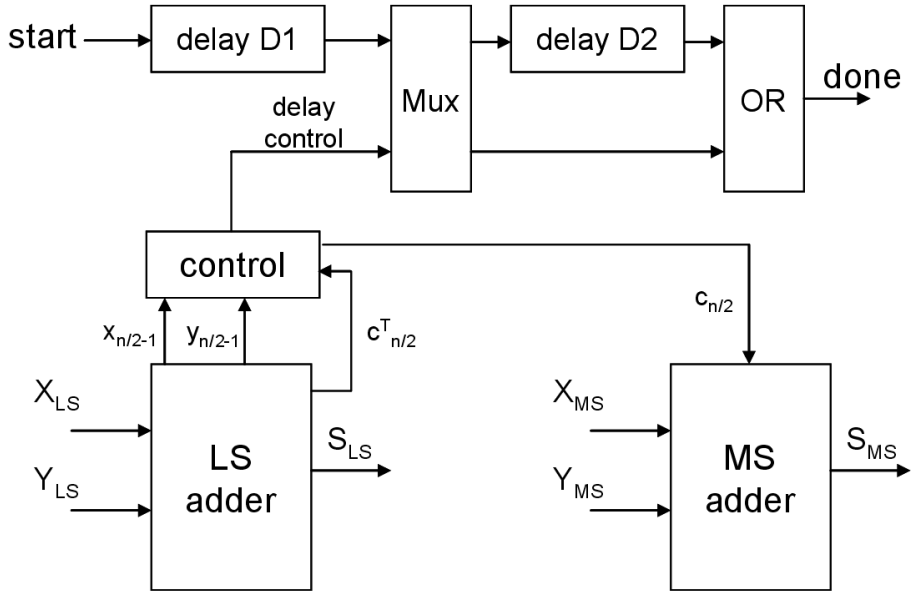
$$\begin{aligned} c_4 &= G_0 + c_0 P_0 \\ c_8 &= G_1 + G_0 P_1 + c_0 P_0 P_1 \\ c_{12} &= G_2 + G_1 P_2 + G_0 P_1 P_2 + c_0 P_0 P_1 P_2 \\ c_{16} &= G_3 + G_2 P_3 + G_1 P_2 P_3 + G_0 P_1 P_2 P_3 + c_0 P_0 P_1 P_2 P_3 \end{aligned} \tag{2.12}$$

Figure 2.4 shows the implemenation of a classic 4-bits CLA, with a lookahead level, and a 16-bits CLA, with 2 levels of lookahead. Nevertheless more complex and faster designs, known as *prefix* adders, can be found in literature, such as [KS73, LF80, Lin81, BK82, HC87], that take the carry lookahead principle to the extreme. Without going deeper into them and repeating the lookahead-tree structure shown in figure 2.4b, it is easy to deduce a delay $\mathbf{O(\log(n))}$. The counterpart of this great performance is the large amount of area required to implement the lookahead levels, which makes CLAs not good for area or power constrained designs.

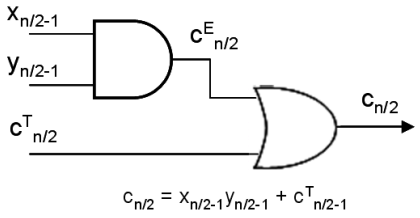
2.1.4. Estimated Carry Adders

The *Estimated Carry Adder* (ESTC) [WDH01, ADH05] is an speculative adder that uses both carry select and lookahead techniques. On the one hand it divides the adder into several modules as the CSEL, but without replicating the most significant modules because it will only make the calculations with one carry. On the other hand, this carry is estimated with the most significant bits from the previous module. If the estimation is correct the execution time will be half time of an adder with the same width. If not, the execution time will be roughly the same as if the modules would have been connected with the ripple carry technique. See figure 2.5. This estimation resembles to the lookahead of the CLAs, due to in most of the cases the carry is only being anticipated. Note that this technique, as the aforementioned ones, allows to use different kinds of adders for the internal modules that compose the adder. For example, two CLA modules can be used as basic blocks and connected with the ESTC technique. However, in this case time analysis would be different. In general if logarithmic-like adders are used as basic blocks and connected with the ESTC technique, delay is *only* reduced from *log(n)* to *log(n/2)*, i.e. *log(n)-1*. This is *only* one level, which in terms of delay could seem a slight reduction, but in terms of area the difference is important due to the large amount of logic needed to forward the carries.

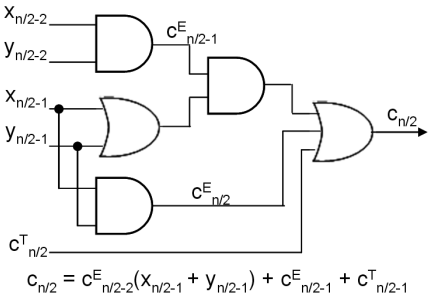
The most interesting feature of this design is shown when the carry estimation fails. In this case the correction of the result is made over the same most significant module, so there is no need to replicate it. The hardware overhead is due to the carry estimation and the control and delay logic for



(a) n-bits ESTC Adder structure



(b) 1-bit carry estimator



(c) 2-bits carry estimator

Figure 2.5: ESTC implementations

generating the *done* signal. Besides, the datapath controller must generate the *start* pulse. See figure 2.5. In this way, ESTCs have an area and consumption a little bit higher than the original associated adder, with an average performance a little bit lower than if they had been built with a pure carry select technique, due to the saving of the replicated most significant modules and associated multiplexers, as explained in subsection 2.1.2.

The ESTCs behavior is asynchronous. One of the advantages of the asynchronous modules is that the execution time can be considered as the average delay, instead of the worst case delay as in the synchronous modules. Therefore, the objective is to be as close as possible to the best case delay so that the average delay is similar. So the percentage of hits in the carry estimation becomes critical. However, most circuits are synchronous today, which

actually converts the asynchronous feature in a considerable disadvantage.

Estimating carries consists in using the most significant bits of the previous module to decide the carry-in for the following module. In this case, authors in [WDH01] utilize the logic **AND** of bits $x_{n/2-1}$ and $y_{n/2-1}$ in order to estimate the next carry-in, i.e. $c_{n/2}$ ($c_{n/2}{}^{E}$). Hence, we will have a 75 % of probability to guess the true carry ($c_{n/2}{}^{T}$), as shown in equation 2.13. That is, we will surely have a carry '1' if both bits are '1', or '0' if both bits are '0'. If they are different, supposing that every case has the same probability, the half of the cases will be guessed. But if $x_{n/2-1} \neq y_{n/2-1}$, as a logic **AND** is being used for estimating, a '0' value will be assumed for $c_{n/2}$, so the true carry will only be guessed if the input carry-in to the *(n/2-1)*th stage is '0'.

$$\begin{aligned} P(c^E_{n/2} = c^T_{n/2}) = P(x_{n/2-1} = y_{n/2-1}) + (P(x_{n/2-1} \neq y_{n/2-1}) * \\ P(c_{n/2-1} =' 0')) = 0{,}5 + (0{,}5 * 0{,}5) = 0{,}75 \end{aligned} \tag{2.13}$$

Authors in [ADH05] apply the same reasoning to increase the hit rate by increasing the number of input bits for estimating the carry, reaching more than 95 % with 4 bits per operand. For example, equation 2.14 calculates the probability of guessing the carry using two input bits per operand, i.e. the corresponding input bits to positions *n/2-1* and *n/2-2*. This calculation consists in unrolling equation 2.13

$$\begin{aligned} P(c^E_{n/2} = c^T_{n/2}) = P(x_{n/2-1} = y_{n/2-1}) + (P(x_{n/2-1} \neq y_{n/2-1}) * \\ P(c^E_{n/2-1} = c^T_{n/2-1})) = P(x_{n/2-1} = y_{n/2-1}) + \\ (P(x_{n/2-1} \neq y_{n/2-1}) * (P(x_{n/2-2} = y_{n/2-2}) \\ + (P(x_{n/2-2} \neq y_{n/2-2}) * P(c_{n/2-2} =' 0'))) \\ = 0{,}5 + (0{,}5 * (0{,}5 + (0{,}5 * 0{,}5))) = 0{,}875 \end{aligned} \tag{2.14}$$

The probabilities calculated in equations 2.13 and 2.14 are the *hit rates* of the estimators. The *failure rates* are then computed as *(1-hit rates)*. Therefore the failure rates progression is 0.25, 0.125, 0.0625, ... if 1, 2, 3, ... input bits from both operands are used. Hence the failure rate expression using k input bits is given by equation 2.15

$$FailureRate(k) = R_{fail}(k) = \frac{1}{2^{k+1}} \tag{2.15}$$

And therefore the hit rate is given by equation 2.16

$$HitRate(k) = R_{hit}(k) = 1 - R_{fail}(k) = 1 - \frac{1}{2^{k+1}} \tag{2.16}$$

The counterpart of increasing the input-bits for estimating carry is the increase in the area, estimation delay and power consumption caused by

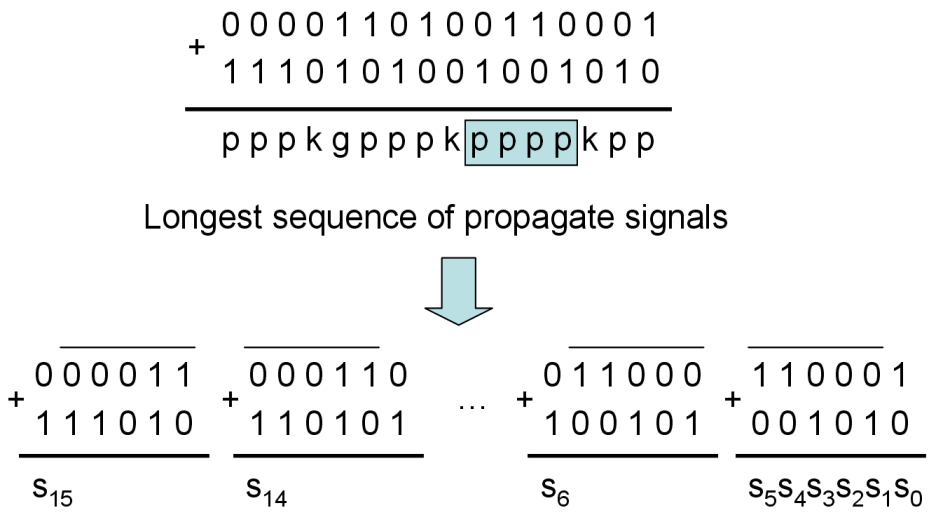


Figure 2.6: 20-bits addition with a longest propagate sequence of k=4

the additional hardware, as in the case of CLAs in section 2.1.3, where the application of lookahead was restricted. Similar forwarding techniques have been also used in [LL00b] for reducing pipeline delay.

To summarize this subsection it can be concluded that the ESTCs can achieve a good performance, similar to the CSELs, but diminishing area overhead. However they operate in asynchronous mode and assume that all the combinations of input bits have the same probability, which does not happen in real circuits [LWS96, LL00a, BKI99].

2.1.5. Carry Lookahead-like Speculative Adders

Carry Lookahead-like Speculative Adders (CLASPs) [VBI08, Cil09] combine the carry lookahead technique with the carry estimation ideas seen in subsection 2.1.4. They are speculative adders able to reduce the typical logarithmic delay from CLAs.

Verma et al. [VBI08] introduced the basis of these designs. The idea consists in dividing the additions into several overlapped groups such that the carry-in to those groups becomes independent from the previous group. Taking into account equations 2.6 and 2.4 it is easy to deduce that c_{i+1} depends on c_i iff p_i='1'. Hence c_i depends on c_{i-1} iff p_{i-1}='1'. In general c_{i+k} is dependant on c_i iff there exists a sequence of k consecutive propagate signals equal to '1' between positions i and $i+k-1$, inclusively. Then c_{i+k+1}, calculated by the stage $i+k$, is independent from c_i. If an oracle provides us the longest sequence of propagate signals, it is possible to execute independently and correctly the summation of the groups.

For example, consider a 20-bits addition and suppose that the longest sequence of propagate signals is k=4. Then c_{i+5} is independent from c_i $\forall$ i, $15 \geq i \geq 0$. See figure 2.6. Note that for every bit in position i, g, p, k mean

$$\begin{aligned} g_i &= x_i \cdot y_i \\ p_i &= x_i + y_i \\ k_i &= \overline{x_i + y_i} \end{aligned} \tag{2.17}$$

Except the first group, every group produces one sum bit, which corresponds to the most significant one inside this group. As the carry-in to the first group is always '0', it can produce correctly so many sum bits as the group length. In the rest of the groups, a carry-in equal to '0' is also supposed, based on the carry independency assumption described in the abovementioned paragraphs. Note that most of the input bits are repeated from one group to the following one, which will produce a great area overhead as counterpart of the great performance. As every group is computed in parallel, the delay would be the same than the sum bits calculation of one group.

The next question to be solved for choosing the correct size of the groups is then ... what is the longest sequence of propagate signals inside every addition? Authors in [VBI08] demonstrate that this sequence is aproximately to *log(n)+12*, if n is the addition width. This happens with a probability equal to 0.9999. However there still exists the probability of failure, so some hardware support for error detecting and error recovery must be included. Every possible chain of *k+1* propagates must be evaluated. Therefore, without going much deeper in the design it is easy to see the large amount of logic that implies, in spite of the great performance, which is better than typical CLAs and even than the prefix adders mentioned in section 2.1.3. In concrete, authors in [VBI08] state that CLASP occupies around 1.5 times the area of a fast logarithmic adder, which is very much. A variation of this design is presented in [Cil09]. It takes into account data correlation, but still suffers the problem of the area overhead. So, in conclusion, these designs are not suitable for an area constrained circuit.

2.1.6. Predictive Adders

In this subsection a new adder structure which addresses the problems presented in subsection 2.1.4 is presented. *Predictive Adders* (PRADDs) [BMM+08a, BMM+08c] are speculative adders that use a similar structure to the ESTCs one. The adder is divided into two halves without replicating the most significant module, such as the ESTC. A predictive module is used in order to operate both addition parts in parallel, instead of the combinational logic for estimating/forwarding it. The main advantage is that PRADDs work in synchronous mode, as the majority of today's circuits.

The proposed prediction technique consists in the use of branch predictors inside FUs to increase their performance without compromising their area. This idea arises from the fact that applications present data correlation [MLS96, SC97], which provides a good scenario for reaching high hit rates. Moreover, predicting a carry is to decide whether a bit is '1' or '0'. Hence this prediction will be somewhat similar to the decision of *taken* or *not taken* in a branch.

Large prediction structures obviously produce large hardware overheads. This disadvantage has motivated the decision of using simple predictors. The following predictors have been considered [HP07, BMM+08a, BMM+08c]:

(1) *One bit predictor*. This is the simplest one. It can be implemented only with a D-flipflop. The predicted carry will be the last true carry.

(2) *Two bits predictor* or *bimodal predictor*. It consists of a finite state machine with four states. In a branch context, "00" and "01" states mean *strongly not taken*, and *not taken*, while "10" and "11" mean *taken* and *strongly taken*. In other words, from the point of view of the carry-out prediction, a '0' value will be predicted for states "00" and "01", and a '1' value for the other states.

(3) *History predictor*. This predictor decides the following carry-out depending on a fixed number of last true carries, after applying some function over these bits. In this case, 3 bits of carry history have been chosen. The decision function will be the majority function. For example, if the last three carries are '0', '1', '0', a '0' value will be predicted.

(4) *Contextual predictor*. This predictor decides the next carry-out according to carry patterns and some history bits. In this case two bits of carry history have been used. For example, if the last two carries are '0' and '0', the pattern will be "00". Therefore, the next carry-out will be the last true carry that happened for the pattern "00".

In order to motivate the use of these predictive structures versus the estimated values via combinational logic, a sequence of several additions taken from the execution of the *Adaptive Differential Pulse Code Modulation* (ADPCM) with real data have been studied in table 2.1. A 16-bits ESTC has executed these additions using 1-bit and 2-bits estimators, as explained in section 2.1.4. A similar adder, but with a 1-bit predictor instead of the estimator, has been utilized too for comparing results. In other words, two 8-bits RCAs have been connected with the 1-bit predictor. In this case, an initial '0' prediction has been supposed.

First column of table 2.1 depicts both 16-bits operands. Columns 2 and 3 identify the two most significant bits from both operands. Columns 4, 6 and 8 are the carry estimation/prediction for the middle bit, i.e. C_7, according

to the corresponding estimator/predictor, while columns 5, 7 and 9 are the true values after executing the additions.

As it can be observed, two failures happen when utilizing both estimators. On the contrary, only one failure is produced when using the predictor. Furthermore, this failure corresponds with the initial prediction. So why is a single 1-bit predictor so accurate ? The answer to this question lies in data correlation. In signal processing and in multimedia applications it is quite common to find values that usually belong to a small interval. Hence, these values are prone to be repeated, or at least to be very similar. And therefore, this data correlation will produce similar carries when adding the operands.

The problem of estimating/predicting carries arises when the estimation/prediction fails. So in order to incorporate these modules to a synchronous context, the worst case delay must be considered. To overcome this limitation the operation delay concept must be changed, that is, the same kind of operation can take a different number of cycles. For example, an adder can be divided into two halves that operate independently. So the cycle time will be roughly the 50 % of the original. If the carry is guessed, every addition executed in that adder will last one cycle, if not, the prediction will be changed for the next cycle, in which the correct result will be calculated with the correct carry. The same idea can be applied to different types of operations. Note that the assumption of the 50 % time reduction is only valid for linear adders. As in the case of ESTCs, logarithmic-like basic blocks interconnected with the prediction technique would reduce 1 forwarding level, which in terms of delay is not too much, but in terms of area is significant.

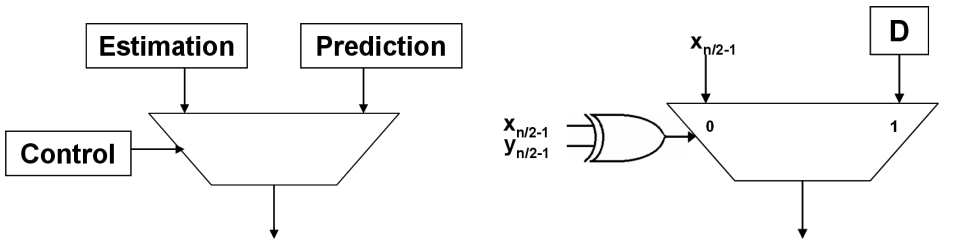
Therefore, the only area overhead consists in deciding when there is a hit or a failure while predicting the carry, and in this way delay elements are not necessary for generating the *Done* signal as in figure 2.5. Moreover, the detection and correction hardware of this kind of adders is much simpler than the evaluation of every possible chain of *k+1* propagates signals, implemented in the CLASPs.

2.1.6.1. Improvement of the hit rate

In the aforementioned [WDH01, ADH05] techniques there is not pure prediction because they use some operand bits and some logic for obtaining a certain value for the carry-out. Hence, this is similar to the forwarding technique in CLAs. The difference is that for some input bits this forwarding is incorrect. On the other hand, the technique of predicting carries is very accurate, but it can mispredict with some combinations of bits that are easily predictable. The key question is why we are going to predict a carry when we are completely sure about its value in an easy way. Therefore the estimation and the prediction techniques can be combined, and use estimation when we are sure of guessing the carry and prediction when we are not. These schemes have been named *hybrid predictors*. See figure 2.7.

Addition	7^{th}-6^{th} bits		1-bit Estimator		2-bits Estimator		1-bit Predictor	
	1^{st} op	2^{nd} op	$C_7{}^E$	$C_7{}^T$	$C_7{}^E$	$C_7{}^T$	$C_7{}^E$	$C_7{}^T$
0000001010101010 + 0011111101110110	10	11	1	1	1	1	0	0
0000001010101111 + 0000000101000001	10	01	0	1	0	1	0	1
0000001010101101 + 0011111101110011	10	11	1	1	1	1	0	1
0000001010110010 + 0000000100010010	10	01	0	1	0	1	0	1
0000001010101111 + 0011111101110001	10	11	1	1	1	1	0	1

Table 2.1: Estimation versus Prediction in the ADPCM decoder



(a) Generic Hybrid Prediction scheme (b) Hybrid predictor implemented with 1-bit Estimator and 1-bit Predictor

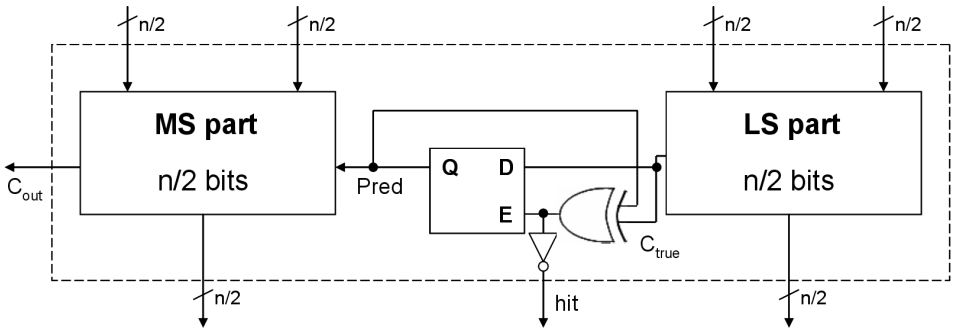
Figure 2.7: Hybrid Carry Predictors

In the example implementation, shown in figure 2.7b, a 1-bit estimator and a 1-bit predictor have been used. That is, if the most significant operands bits are equal, the carry will surely be '0' or '1'. When these bits are different the carry-out is uncertain, so the predictor value will be utilized. Note that the estimator is only used when both bits are equal, so the estimator can be reduced to a forwarding of the bit used for estimation.

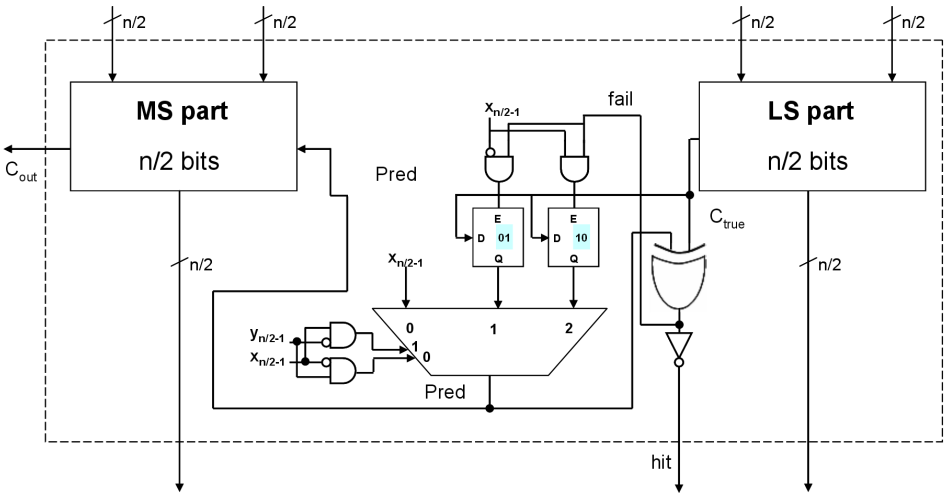
Another example of hybrid predictor is the *1-Bit Input Pattern Predictor* (1BIPP), which appears in [BMM+10, BMM+11]. It consists of a table accessed by the patterns built with certain input bits. This pattern is composed of the most significant bits from the least significant module, i.e. $x_{n/2\text{-}1}$ and $y_{n/2\text{-}1}$. Every entry keeps the last true carry value at the time when this pattern happened. Note that in the case of adders, if x_i and y_i input bits are being used as patterns, and if c_{i+1} is being predicted, as in [BMM+10, BMM+11], only two entries are required, because when $x_i = y_i$ the carry-in is known and can be forwarded in a similar fashion to the hybrid predictor of figure 2.7b.

The increase of the complexity in the predictor design is motivated from the necessity of including several speculative FUs in the design. It is necessary to increase as much as possible the individual hit rate of every predictor, because when several predictors, i.e. speculative FUs, are working at the same time the datapath will infere a global hit iff all the predictors hit. This question will be deeply discussed in chapters 3 and 4.

The design of this kind of adders is simple. Two smaller adders are used. The most significant one takes the carry-in from the predictor. See figure 2.8. Figure 2.8a depicts a PRADD implemented with a 1-bit predictor, which was presented in [BMM+08a, BMM+08c]. The predictor is composed of a D-flipflop whose writing will be enabled everytime the true carry (C_{true}) is different from the predicted one (*Pred*). This will produce a '0' value in the output *hit* signal. Otherwise *hit*='1'. Thus, this *hit* signal will be responsible for indicating to the datapath controller that there is a failure or not in the corresponding PRADD.



(a) Predictive adder with 1-bit predictor



(b) Predictive Adder with 1-Bit Input Pattern Predictor

Figure 2.8: Predictive Adder implementations

On the other hand, in figure 2.8b the selected predictor is the 1BIPP. As well as in the 1-bit predictor, the prediction is updated when it is different from the true carry. The *hit* signal implementation is the same than in the previous case. However, the writing of the predictor is organized differently. If the pattern $x_{n/2\text{-}1}y_{n/2\text{-}1}$ is "01", the true carry must be written in the D-flipflop labeled as "01". Analogously for the pattern "10". Note that when both pattern bits are equal, there is no need to write because the carry-in can be forwarded. Finally, the reading of the prediction is controlled with a multiplexer and certain logic, depending on the pattern bits. Note that this was not necessary with the 1-bit predictor, because the read carry-in always proceeded from the output of the D-flipflop.

Therefore, PRADDs are faster than the non-predictive corresponding adders, and with a really low area overhead, because there is no need to use delay elements and the control logic is quite simple. Moreover, this scheme is general. Whatever kind of adder and whatever kind of predictor can be

Cycle	X	Y	Pattern	$Pred_{val}$	c_2	Z	c_{out}	hit
i	0000	0110	01	0	0	0110	0	1
i+1	0011	0001	10	0	1	0000	0	0
i+2	0011	0001	10	1	1	0100	0	1
i+3	1111	0010	11	1	1	0001	1	1

Table 2.2: Internal and output signals evolution for a three input sequence inside a Predictive Adder implemented with a 1-bit Input Pattern Predictor

combined. However, it will be necessary to be careful with the selection of the predictor, in order to comply with the performance-area tradeoff.

2.1.6.2. Predictive Adders example of use

In this subsection an example of how the PRADDs operate will be presented.

Let's consider the speculative adder shown in figure 2.8b. A sequence of three inputs is studied. The evolution of the internal and output signals is shown in table 2.2. The three leftmost columns represent the cycle and the inputs to the adder; the middle ones are signals inside the adder, namely: the pattern composed with the middle bits of X and Y, the carry prediction and the studied true carry (c_2), respectively; while the rightmost ones are the sum bits, the carry-out and the hit signal, i.e. the output signals of the adder.

Suppose that the initial prediction is '0' for every pattern. Note that as the pattern is composed by x_1 and y_1, and c_2 is being predicted, strictly speaking, prediction will only happen for those combinations such that $x_1 \neq y_1$, because if they are equal the carry will be forwarded, as explained in subsection 2.1.6.1. The first addition is correctly executed after one cycle because the prediction is the same as the carry-out of the least significant part of the adder, i.e. c_2. The second addition produces a failure because the predicted carry is '0' and the true carry is '1'. Then prediction is updated for the following cycle, when the most significant part will be corrected. Finally the last addition of the sequence is performed properly because the predicted value and the true carry are both '1'.

2.2. Multipliers design

Products are one of the most usual operations while designing circuits, so multipliers design is critical in order to satisfy the constraints imposed by the designer. If adders design was already critical because many operators are built with them, multipliers design usually becomes essential too because

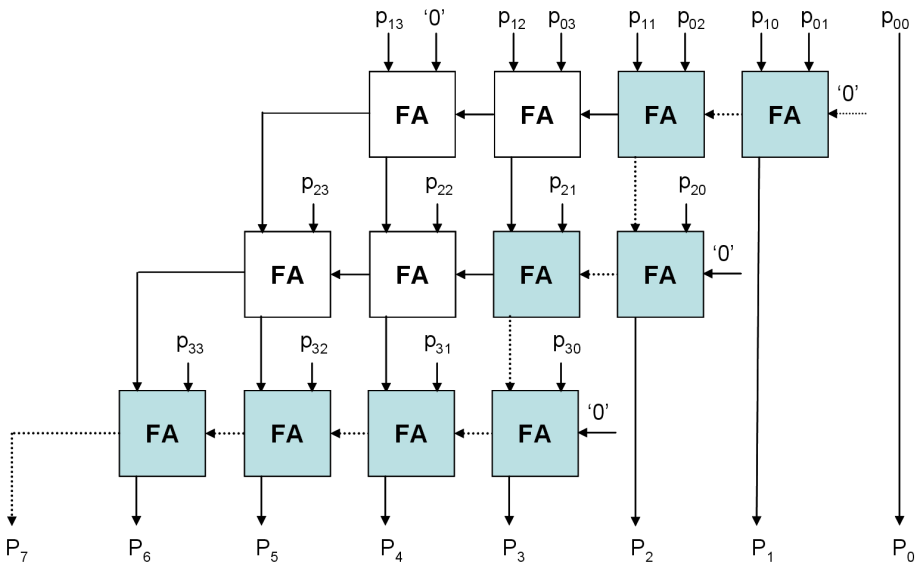


Figure 2.9: 4x4 Ripple Carry Multiplier

they often *dominate* the datapaths. In homogeneous datapaths and many heterogeneous ones, multipliers are the biggest modules inside them, so they contribute very much to overall area, power consumption, etc. And in the case of monocycle implementations, multipliers critical path will also determine the cycle time of the whole circuit.

In this section an overview of classical parallel multipliers will be given and a new speculative design, which will be used later in this Ph.D. Thesis, will be described.

2.2.1. Ripple Carry Multipliers

Ripple Carry Multipliers (RCMs) [Kor02] are the most straightforward method to multiply two unsigned numbers. It is the direct mapping of the traditional integer multiplication algorithm in base 10. Supposing that X has m bits and Y has n bits, m, $n \geq 0$, we will say that $X*Y$ is a mxn product. Then given a mxn product such that $X=x_{m-1}x_{m-2} \ldots x_0$ and $Y=y_{n-1}y_{n-2} \ldots y_0$, the traditional algorithm multiplies X per every y_i, $n > i \geq 0$, and adds the partial products after shifting them properly according to their weight. Figure 2.9 depicts the implementation of a typical 4x4 RCM. Note that, as in the case of the adders, the FAs with a constant '0' input can be simplified into HAs.

A mxn RCM is composed by *n-1* rows, each of them with m columns. The critical path is given by the size of its inputs. Then in a mxn RCM the delay will be proportional to $m+n$, as shown by the solid FAs in figure 2.9.

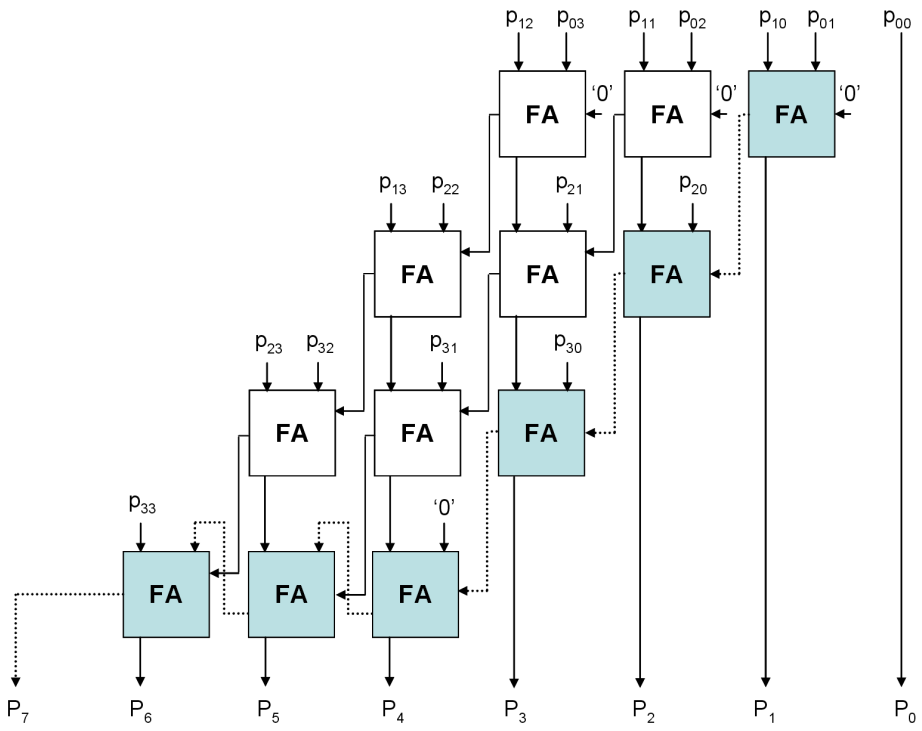


Figure 2.10: 4x4 CSA Multiplier

The input signals that feed the FAs correspond with the logic **AND** of the inputs, i.e. $p_{ij}=x_i \cdot y_j$.

Finally note that RCMs are only able to multiply positive numbers, so if we are working with negative numbers it will be necessary to look for another structure. Besides, as the reader will see in the following subsections, RCM structure is not proper for applying prediction techniques.

2.2.2. Carry Save Multipliers

Carry Save Multipliers (CSAMs) or *Braun* multipliers [Kor02] are multipliers that break the RCMs carry chain in order to accelerate multiplications. A *mxn* multiplier is composed by *n* rows, each of them with *m-1* columns. Figure 2.10 depicts an example of 4x4 CSAM. As in the RCM case, $p_{ij}=x_i \cdot y_j$. Note also that the FAs with '0's inputs can be simplified into HAs.

CSAMs propagate the carry-out of every FA stage to the following row. In this way the multiplier is clearly divided in two regions: the *Partial Product Accumulator*, composed by the *n-1* first rows, and the *Final Stage Adder*, which consists of a two input adder that composes the final result. Therefore the critical path is determined by the adder in the last stage. In a *mxn*

multiplier the delay is equal to the delay of the *n-1* rows plus the delay of the *Final Stage Adder*. For example, if the last stage adder is a m-bits RCA, as in figure 2.10, the overall CSAM delay would be proportional to *m+n-2*.

This *m+n-2* delay corresponds with a linear implementation of the Partial Product Accumulator. Nevertheless, this *Carry Save Adders* (CSAs) array can be organized as a tree structure, such as the Wallace [Wal64] or the Dadda trees [Dad65, Dad76]. Hence, the weight of the Final Stage Adder in the overall delay will increase. Thus, as the tree delay is logarithmic, the overall delay in a CSAM with a CSA tree will be proportional to *log(n)+delay(Last Stage)*.

Therefore, CSAMs allow the use of whatever kind of adder in the last stage. Besides they are slightly faster than the RCMs and present a regular structure, which makes them suitable for the VLSI design. However, the Braun multiplier is not able to use negative numbers, a brief modification is required.

2.2.2.1. Baugh-Wooley Multipliers

The *Baugh-Wooley Multiplier* (BWM) [BW73] is an enhanced version of the Braun multiplier which operates with both possitive and negative numbers. Intuitively, the idea is to make sure that all the input bits to the cells are possitive. Mathematically, taking into account that the decimal values of two 2-Complement n-bits numbers, say X and Y, are

$$\begin{aligned} X &= -X_{n-1}2^{n-1} + \sum_{i=0}^{n-2} X_i 2^i \\ Y &= -Y_{n-1}2^{n-1} + \sum_{j=0}^{n-2} Y_j 2^j \end{aligned} \tag{2.18}$$

the product $P = XY$ can be expressed as

$$\begin{aligned} P &= XY \\ &= X_{n-1}Y_{n-1}2^{2n-2} + \sum_{i=0}^{n-2}\sum_{j=0}^{n-2} X_i Y_j 2^{i+j} - X_{n-1}2^{n-1}\sum_{j=0}^{n-2} Y_j 2^j \\ &- Y_{n-1}2^{n-1}\sum_{i=0}^{n-2} X_i 2^i \end{aligned} \tag{2.19}$$

Then, in order to remove the subtractions, some transformations are

required. See equation 2.20.

$$
\begin{aligned}
-X_{n-1}2^{n-1}\sum_{j=0}^{n-2}Y_j2^j &= 2^{n-1}\left(-\sum_{j=0}^{n-2}Y_jX_{n-1}2^j\right)\\
&= 2^{n-1}\left(2^n+2^{n-1}+\overline{X}_{n-1}2^{n-1}+X_{n-1}+\sum_{j=0}^{n-2}\overline{Y}_jX_{n-1}2^j\right)\\
-Y_{n-1}2^{n-1}\sum_{i=0}^{n-2}X_i2^i &= 2^{n-1}\left(-\sum_{i=0}^{n-2}X_iY_{n-1}2^i\right)\\
&= 2^{n-1}\left(2^n+2^{n-1}+\overline{Y}_{n-1}2^{n-1}+Y_{n-1}+\sum_{i=0}^{n-2}\overline{X}_iY_{n-1}2^i\right)
\end{aligned}
\tag{2.20}
$$

Equation 2.20 is easily verifiable thanks to the following lemma.

Lemma 1. *Let U=(0,0,$u_{n-2}k$,$u_{n-3}k$,..., u_0k), V=(0,$\overline{k}$,$\overline{u}_{n-2}k$,$\overline{u}_{n-3}k$,..., $\overline{u}_0k$) and W=(1,1,0,0,..., 0,k) be three (n+1)-bits 2-Complement numbers, k ∈ {0,1}. Then,*

$$-U = V + W \quad \forall\ k$$

Proof. There are two cases:

(1) k=0 ⇒ U = -U =(0,0,0,...,0), V=(0,1,0,...,0) and W=(1,1,0,...,0)
V + W = (0,0,0,...,0) with c_{out}='1'. Note that this c_{out} has a weight 2^{n+1}. As in equation 2.20 it would be multiplied by the constant 2^{n-1}. Hence it would have an actual weight of 2^{2n}, which can be ignored due to the nature of the 2-Complement addition.

(2) k=1 ⇒ U = (0,0,u_{n-2},u_{n-3},...,0), V=(0,0,$\overline{u}_{n-2}$,$\overline{u}_{n-3}$,...,$\overline{u}_0$) and W=(1,1,0,0,...,1)
V + W = (0,0,$\overline{u}_{n-2}$,$\overline{u}_{n-3}$,...,$\overline{u}_0$) + (1,1,0,0,...,1) = (1,1,$\overline{u}_{n-2}$,$\overline{u}_{n-3}$,...,$\overline{u}_0$) + (0,0,0,0,...,1) = C1(U) + 1 = C2(U) = -U where C1(U) and C2(U) are the 1-Complement and 2-Complement of U.

□

Equation 2.21 depicts how to apply the lemma to equation 2.20, selecting

the vectors U, V and W, and the value of k.

$$
\begin{aligned}
-X_{n-1}2^{n-1}\sum_{j=0}^{n-2}Y_j2^j &= 2^{n-1}\left(\underbrace{-\sum_{j=0}^{n-2}Y_j\overbrace{X_{n-1}}^{k}2^j}_{U}\right)\\
&= 2^{n-1}\left(\underbrace{\overline{X}_{n-1}2^{n-1}+\sum_{j=0}^{n-2}\overline{Y}_jX_{n-1}2^j}_{V}+\underbrace{2^n+2^{n-1}+X_{n-1}}_{W}\right)\\
-Y_{n-1}2^{n-1}\sum_{i=0}^{n-2}X_i2^i &= 2^{n-1}\left(\underbrace{-\sum_{i=0}^{n-2}X_i\overbrace{Y_{n-1}}^{k}2^i}_{U}\right)\\
&= 2^{n-1}\left(\underbrace{\overline{Y}_{n-1}2^{n-1}+\sum_{i=0}^{n-2}\overline{X}_iY_{n-1}2^i}_{V}+\underbrace{2^n+2^{n-1}+Y_{n-1}}_{W}\right)
\end{aligned}
\tag{2.21}
$$

Therefore, applying the lemma, the substractions can be substituted by

additions, as shown by equation 2.22.

$$
\begin{aligned}
&-X_{n-1}2^{n-1}\sum_{j=0}^{n-2}Y_j2^j - Y_{n-1}2^{n-1}\sum_{i=0}^{n-2}X_i2^i = \\
&\quad = 2^{n-1}\left(2^n + 2^{n-1} + \overline{X}_{n-1}2^{n-1} + X_{n-1} + \sum_{j=0}^{n-2}\overline{Y}_jX_{n-1}2^j\right) \\
&\quad + 2^{n-1}\left(2^n + 2^{n-1} + \overline{Y}_{n-1}2^{n-1} + Y_{n-1} + \sum_{i=0}^{n-2}\overline{X}_iY_{n-1}2^i\right) \\
&\quad = 2^{2n} + 2^{2n-1} + (\overline{X}_{n-1} + \overline{Y}_{n-1})2^{2n-2} + (X_{n-1} + Y_{n-1})2^{n-1} \\
&\quad + \sum_{j=0}^{n-2}\overline{Y}_jX_{n-1}2^{j+n-1} + \sum_{i=0}^{n-2}\overline{X}_iY_{n-1}2^{i+n-1}
\end{aligned}
$$

2^{2n} can be ignored because of the 2-Complement addition nature.

$$
\begin{aligned}
&\quad = 2^{2n-1} + (\overline{X}_{n-1} + \overline{Y}_{n-1})2^{2n-2} + (X_{n-1} + Y_{n-1})2^{n-1} \\
&\quad + \sum_{j=0}^{n-2}\overline{Y}_jX_{n-1}2^{j+n-1} + \sum_{i=0}^{n-2}\overline{X}_iY_{n-1}2^{i+n-1}
\end{aligned}
\tag{2.22}
$$

Substituting equation 2.22 in equation 2.19, the following expression is obtained

$$
\begin{aligned}
P &= XY \\
&= X_{n-1}Y_{n-1}2^{2n-2} + \sum_{i=0}^{n-2}\sum_{j=0}^{n-2}X_iY_j2^{i+j} \\
&+ 2^{2n-1} + (\overline{X}_{n-1} + \overline{Y}_{n-1})2^{2n-2} + (X_{n-1} + Y_{n-1})2^{n-1} \\
&+ \sum_{j=0}^{n-2}\overline{Y}_jX_{n-1}2^{j+n-1} + \sum_{i=0}^{n-2}\overline{X}_iY_{n-1}2^{i+n-1} \\
&= \sum_{i=0}^{n-2}\sum_{j=0}^{n-2}X_iY_j2^{i+j} + \sum_{j=0}^{n-2}\overline{Y}_jX_{n-1}2^{j+n-1} + \sum_{i=0}^{n-2}\overline{X}_iY_{n-1}2^{i+n-1} \\
&+ 2^{2n-1} + (\overline{X}_{n-1} + \overline{Y}_{n-1} + X_{n-1}Y_{n-1})2^{2n-2} + (X_{n-1} + Y_{n-1})2^{n-1}
\end{aligned}
\tag{2.23}
$$

The expression given by equation 2.23 only contains additions and is easily mappable onto a CSA-like cells matrix. Note that the general proof for two generic m and n bits numbers is given in detail in [BW73]. Figure

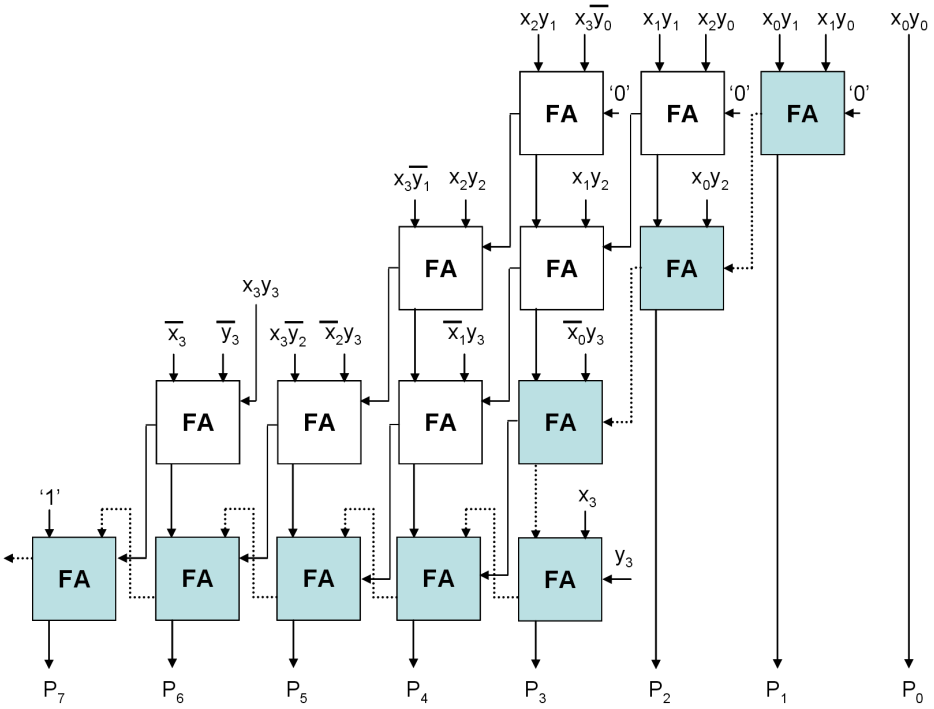


Figure 2.11: 4x4 Baugh-Wooley Multiplier

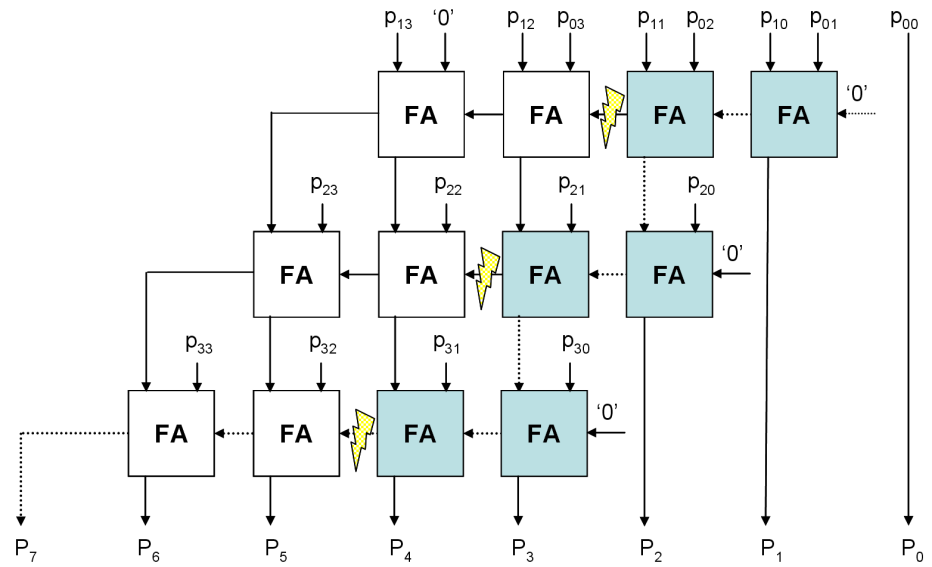
2.11 shows the structure of a 4x4 BWM. As it is observed, the FAs array maps directly equation 2.23. Note that the last row, as in the CSAM case, is where the final result is composed. Concretely in figure 2.11, the final stage has been implemented with a RCA.

With a RCA in the last stage the delay of the BWM is proportional to $m+n$, as depicted by the solid blocks of figure 2.11. The reader may note that in comparison to the CSAM the delay has been increased, from $m+n-2$ to $m+n$. But this increase is negligible in exchange for the possibility of using negative numbers.

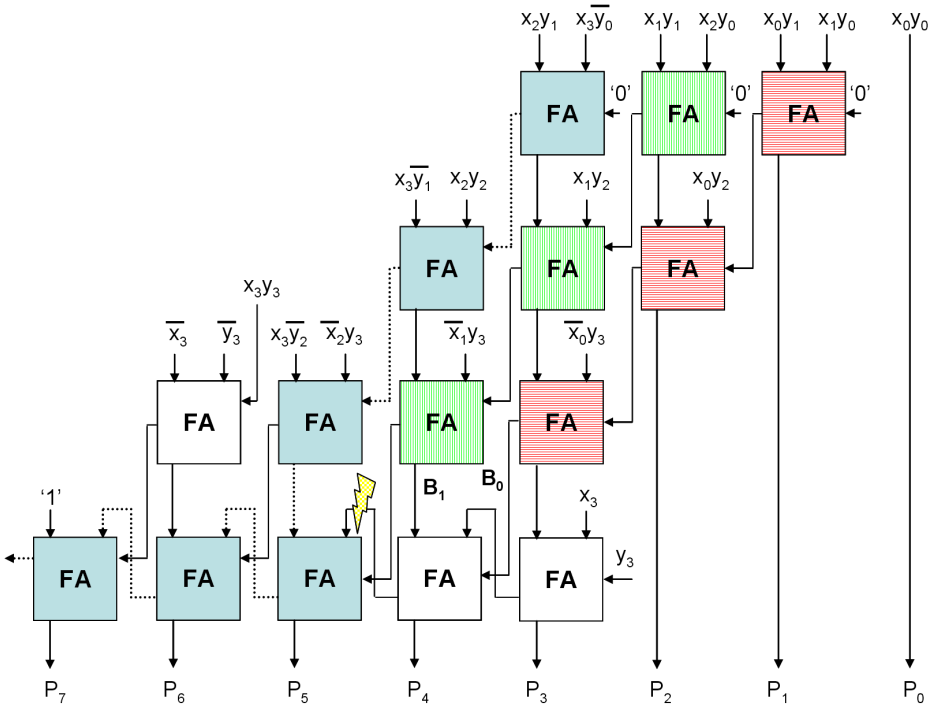
Another structures able to multiply both possitive and negative numbers are [Pez71] and [Boo51]. In [Pez71], Pezaris develops a CSAM-like multiplier, but with an irregular structure, while in [Boo51] Booth applies recodification to the inputs of the multiplier, and it is not interesting from the point of view of this Ph.D. Thesis.

2.2.3. Predictive Multipliers

Predictive Multipliers (PRMs) [BMM+08c] are parallel multipliers that try to reduce their critical path by means of predicting some internal signals, as the PRADDs presented in subsection 2.1.6. PRMs presented in this Ph.D.



(a) Breaking critical path in a Ripple Carry Multiplier



(b) Breaking critical path in a Baugh-Wooley Multiplier

Figure 2.12: Breaking critical path in parallel multipliers

Thesis will have a CSAM-like structure with a Predictive Adders, like the one described in section 2.1.6 in the last stage. As in this Ph.D. Thesis negative

numbers will be considered too, the concrete implementation will consist of a Baugh-Wooley Multiplier with a Predictive Adder in the last stage. In this way, in a mxn PRMs the critical path is diminished from $m+n$ to $\lceil m/2 \rceil + n$, achieving a 25 % reduction in execution time if $m \approx n$, approximately. Moreover, it must be taken into account that if the CSA structure were implemented with a tree structure [Wal64, Dad65, Dad76], the gain would be even greater.

Note that this predictive behavior is not possible with the classical RCMs. In RCM structures in order to obtain the same performance it is necessary to break the carry path by inserting a predictor in every row, while in the BWM ones it is enough to insert only one predictor in the last RCA stage. Figure 2.12 shows where to break the carry path in order to diminish critical path in a 25 % for both RCMs and BWMs. Thunder signs show where to insert the predictors in order to achieve the desired performance improvement, while solid boxes indicate the critical path.

Obviously, RCM implementations oblige us to insert a predictor per row, as shown in figure 2.12a, which is impractical with more than 3 or 4 rows. On the one hand because of the area overhead, and on the other because having so many predictors diminishes rapidly the probability of guessing all the carries for executing the product in *low-latency* mode. Hence, a CSAM-like implementation is the best choice, because for whatever mxn size only one predictor is required.

Finally, note the horizontally red and vertically green stripped boxes in figure 2.12b, which point out the critical path to generate B_0 and B_1, respectively. These bits are the ones that would compose the pattern B_1B_0, in case that a 1-Bit Input Pattern Predictor were being utilized, as in the Predictive Multipliers used in the experiments of this Ph.D. Thesis. It is easy to verify that these pattern bits will be ready when necessary, i.e. after *n-1* FAs.

2.2.3.1. Predictive Multipliers example of use

Predictive Multipliers are Baugh-Wooley Multipliers with a speculative adder in their last stage. Hence, critical path is reduced to a half in this last stage and the overall reduction is 25 %, taking into account a whole nxn multiplier with a linear implementation of the partial product matrix. Let's consider then a PRM as the one illustrated in figure 2.12b, with a last stage implemented with a PRADD as the one depicted in figure 2.8b. The reader must remember that the utilized predictor was the 1BIPP, i.e. a table accessed with a pattern composed of the middle bits of the operands. However this table was simplified, because when both middle operand bits were equal, the predicted carry could be directly forwarded.

A three inputs sequence is studied in table 2.3. A general scheme of the multiplier, with the corresponding patterns of these sequence, can be

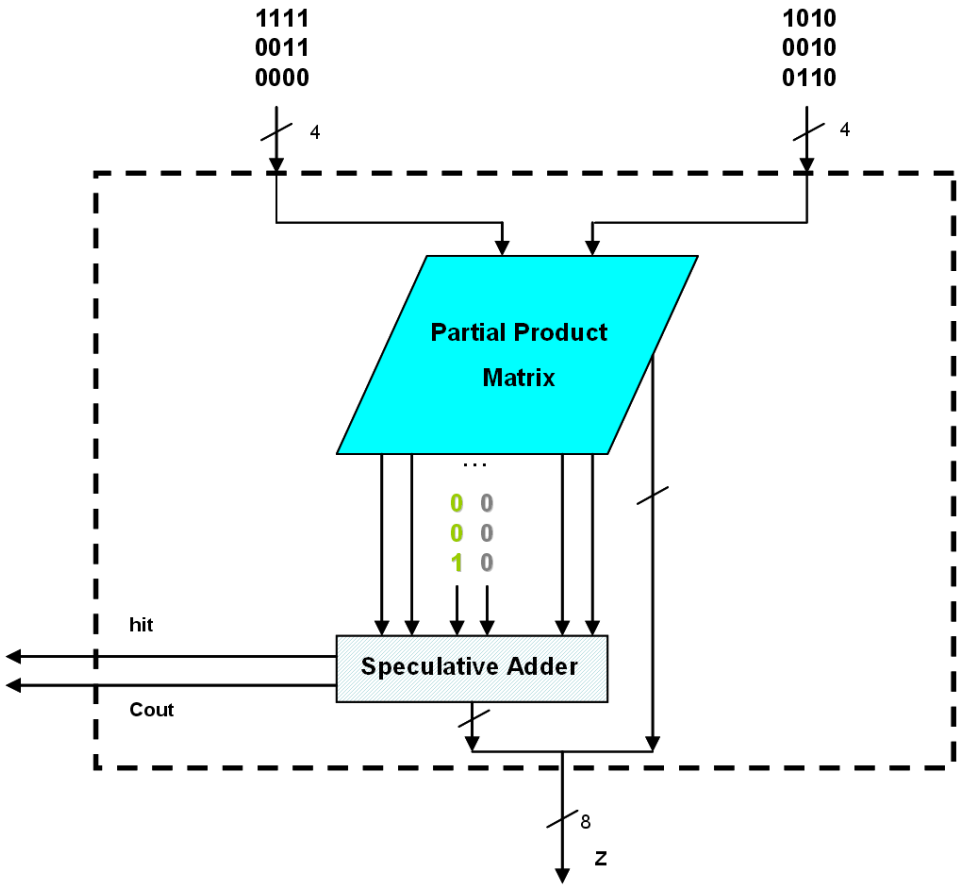


Figure 2.13: Predictive Multiplier structure

Cycle	X	Y	Pattern	$Pred_{val}$	$Cin_{n/2}$	Z	Cout	hit
i	0000	0110	00	0	0	00000000	1	1
i+1	0011	0010	00	0	0	00000110	0	1
i+2	1111	1010	10	1	0	11100110	1	0
i+3	1111	1010	10	1	1	00000110	1	1

Table 2.3: Internal and output signals evolution for a three input sequence inside a Predictive Multiplier implemented with a 1-bit Adaptive Predictor in the middle of the final stage adder

seen in figure 2.13. Note that the patterns are composed of the middle bits of the PRADD operands. This PRADD is located in the last stage of the multiplier. The initial predictions are '0' for every pattern (i.e. "10" and "01", in the other cases the carry prediction consists in a forwarding). Under these

conditions the first two products guess the carry but not the third. Therefore the predicted carry for pattern "10" is updated and one more cycle is spent on correcting the result with the new value for the predicted carry.

2.3. Time Modeling of FUs

This section will perform a timing analysis and modeling of the FUs that will be utilized in this Ph.D. Thesis.

First, consider a RCA. If a predictor is introduced for speculating the carry bit in the middle of the carry chain, the RCA can work twice as fast. This is the case of the PRADD explained in subsection 2.1.6. Thus, consider an n-bits PRADD. If the $n/2$-bits delay constitutes a time unit (i.e., cycle), then, the PRADD latency can be modeled as 1 cycle if the prediction is a hit, and 2 cycles if it is a miss. Hence, the RCA latency will be modeled as two cycles.

For the case of the multiplier, consider the PRM described in subsection 2.2.3. The PRADD located in the last stage of the multiplier will take 1 time unit if it hits, and 2 time units if it does not. The carry save propagation has a depth of n bits, which is modeled as two cycles. Therefore, an nxn PRM will take 3 cycles if prediction is correct and 4 if it is not. The BWM, on the contrary, i.e. the corresponding non-speculative FU, will always take 4 cycles.

In the case of CSELs structures the time analysis is the same than with PRADDs and PRMs if there are no failures, because critical path is reduced as in the speculative modules. The difference is that CSEL structures need more logic to perform this critical path decrease. Note that in the case of multipliers, the design will be a BWM too, but with a CSEL last stage, instead of the RCA utilized in the common BWM implementation.

Logarithmic FUs have also been modeled based on the speculative adder described in subsection 2.1.5. This adder takes 1 cycle if it hits and 2 if it misses in the prediction. However, the adder delay is $\mathbf{O(\log(n))}$. Hence, if the previous time unit, i.e. $n/2$, is utilized, it will be difficult to obtain an integer value for the adder latency. Thereby, in the analysis with logarithmic modules, $log(n)$ will be the time unit.

The speculative multiplier is composed by a BWM, but with the speculative logarithmic adder in the last stage. It must be taken into account that the carry save structure delay is $\mathbf{O(n)}$ and the adder delay is $\mathbf{O(\log(n))}$. Thus, the last stage adder latency will last 1 cycle if it hits, and two if it does not, while the number of required cycles for the carry save part will be calculated as $\lceil n/log(n) \rceil$. As in the experiments performed in chapters 3 and 4 the data width is 16, the carry save part will be computed in 4 cycles. So overall this logarithmic multiplier structure wil take 5 cycles if it hits and 6 if it misses. Then, the corresponding non-speculative adders and multipliers

(a) F1

(b) Chess

(c) Universe

(d) Physicists: Hawking, Newton and Einstein

Figure 2.14: JPEG2000 decoder input photos

will take 2 and 6 cycles, respectively.

This analysis has been made considering multicyle FUs. In the monocycle case every FU will take 1 cycle if it hits and 2 cycles if it does not. Also, note that for every considered speculative design, the worst case latency is identical to the latency of the corresponding non-speculative design.

2.4. Experimental Results

The results related to the Predictive FUs that have been described in the aforementioned sections 2.1.6 and 2.2.3 will be evaluated with several experiments [BMM+08a, BMM+08c].

Firstly the data given by a set of additions proceeding from the simulation of the ADPCM decoder [TC90] and the *Discrete Cosine Transform* (DCT) in the JPEG2000 decoder [Cha99] have been collected, and the hit percentage of the aforementioned estimators/predictors has been calculated. 16-bits adders composed by two 8-bits modules have been used in the ADPCM, while 32-bits adders built of two 16-bits modules have been chosen for the JPEG2000. In both cases a structure similar to the PRADD shown in subsection 2.1.6 has been supposed. The reference example *clinton.adpcm* [Jan01] has been

the input in the ADPCM, while in the JPEG2000 four photos, shown in figure 2.14, have been utilized.

These results are shown in table 2.4a. The leftmost column indicates the kind of estimator or predictor. The first three ones are estimators with 1, 2, 3 input bits like the ones explained in [WDH01, ADH05]. The following four ones are predictors as the aforementioned ones in subsection 2.1.6. The four bottonmost predictors are hybrid structures such as the ones explained in subsection 2.1.6.1. Note that these hybrid predictors have been built with a 1-bit estimator and the pure associated predictor. For example, the *Hybrid2BPPred* is a hybrid predictor composed of a 1-bit estimator and a 2-bits predictor. The last row shows the average results.

With the percentages shown in table 2.4a it can be asserted that pure predictors are as accurate as a 2-bits estimator, and hybrid predictors as a 3-bits estimator.

In order to introduce estimators in the synchronous context, some logic must substitute the controller logic shown in figure 2.5 to store and select the corrected carry for the next cycle, for example a D-flipflop and some extra logic. Hence, the final estimator synchronous structure should actually be like a hybrid predictor composed of the corresponding estimator (1-bit, 2-bits, etc.) and a 1-bit predictor. Besides some extra logic will be required for controlling the selection of the estimator or the D-flipflop (after a failure). Therefore the area penalty will be lower in the case of pure predictors. Besides as the hit rates are even higher for some simpler pure predictors, it can be concluded then that prediction behaves is at least so good as estimation.

Table 2.4b shows the hit percentages reached with the ADPCM products, supposing that *fmult1* and *mix1* multipliers consists of a 6x6 and a 12x8 PRMs, respectively. Table 2.4c shows the hit percentages obtained with the DCT products in the JPEG2000, supposing that 16x16 PRMs are used. This study has been made with 4 different inputs, reaching in every case almost 99 % or higher hit percentages. Note that the PRMs utilized in these experiments utilize a 1-bit predictor.

In order to measure delay reduction and area overhead using predictive techniques, a RCA and a BWM have been synthesized with the commercial tool *Synopsys Design Compiler*. The target library used for both cases is *VTVTLIB25* by Virginia Tech. based on a 0,25 μm TSMC technology. The results are shown in table 2.5. Columns 2, 3 and 5, 6 depict the delay and area for both common and predictive designs. In this case, predictive designs use the 1-bit predictor. Columns 4 and 7 show the delay and area overhead, respectively. A negative percentage means a reduction and a positive one means an increase. Delay is measured in picoseconds (ps) and area in μm^2.

The RCA and BWM have been synthesized for several sizes. In the RCA implementation the delay gain with prediction is greater than 40 % and increases as the size of the adder is augmented, approaching to the theoretical

Estimator/Predictor	ADPCM	JPEG
1-bit EST	91.8	98.9
2-bits EST	94.2	98.9
3-bits EST	96.4	99.0
1-bit Pred	94.3	97.8
2-bits Pred	94.8	98.8
2-bits Contextual	94.2	97.8
3-bits History	94.6	98.9
Hybrid 1BPred	96.2	98.1
Hybrid 2BPred	96.1	99.6
Hybrid 2BContextual	96.0	98.1
Hybrid 3BHistory	96.0	99.5
Average	95.0	98.5

(a) Hit percentage from ADPCM decoder and JPEG2000 additions set

Module	%Hit
fmult1	88.81
mix1	99.25
Average	94.03

(b) Hit percentage from products in the ADPCM decoder

Input	%Hit
F1	99.24
Chess	99.83
Universe	99.85
Physicists	99.56
Average	99.37

(c) Hit percentage from products in the DCT with four different inputs

Table 2.4: Hit percentages

FU	Delay	Delay Pred	%Pen	Area	Area Pred	%Pen
8-bits RCA	3486	2050	-41.19	6564	7332	11.70
16-bits RCA	6931	3773	-45.56	12966	13619	5.04
32-bits RCA	13822	7218	-47.78	25376	26421	4.12
8x8 BWM	8385	6890	-17.83	43371	44004	1.46
16x16 BWM	17790	14026	-21.16	175511	176445	0.53
32x32 BWM	36833	28523	-22.56	683898	685103	0.17

Table 2.5: RCA and BWM delay and area with and without predictive techniques

limit which is the 50 % of gain. The same fact happens with the Braun multiplier, but in this case the initial gain is 18 % approaching to the theoretical limit of 25 % delay gain.

In terms of area, the decrease in the overhead percentage with respect to the inputs size is quite logical, because always one predictor is being used, whatever the operator dimensions.

Chapter 3

Centralized Management of SFUs in HLS

Everything must be made as simple as possible. But not simpler.

Albert Einstein

Centralized Management (CenM) [BMM+10, BMM+11] is the most straightforward method to include Speculative Functional Units in the High-Level Synthesis flow. The problem of using SFUs is to answer the question: *what happens if there is a failure?* Centralized Management does nothing special, it simply stops the datapath and prepares it for making the correction in the next cycle. In other words, this approach consists in indicating to the controller that the datapath cannot make a transition to the next state whenever there is a failure in any SFU predictor.

It is important to note that results will be written iff the *cycle hit signal* is true. If the cycle hit signal is false, i.e. at least there is one SFU hit signal with false value, none of the results can be written, because it would be possible to overwrite a register that will be used in the succeeding correction cycle. For example, consider R1 = R1 + R2, if there were a failure and R1 were written, R1 would be corrupt in the next cycle.

In the next sections, the execution paradigm which is used with Centralized Management, the architecture, an example of use and some results will be described and discussed. Predictive adders and multipliers from chapter 2 will be used. Firstly, a monocycle implementation will be supposed for introducing the basis of Centralized Management. Afterwards multicycle and chaining adaptation will be explained too.

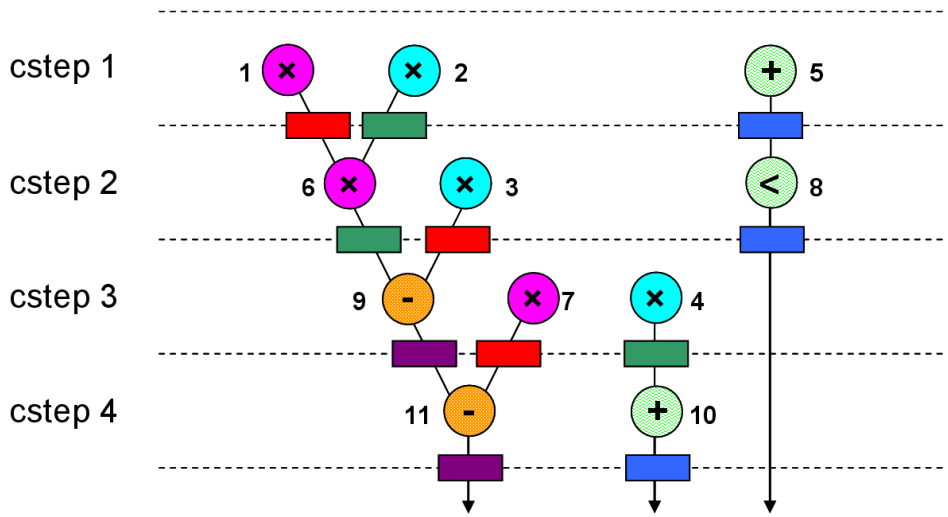


Figure 3.1: DiffEq scheduling and FU and register binding

cstep	M1	M2	A1	A2	R1	R2	R3	R4
1	1	2	5		1	2	5	
2	6	3	8		3	6	8	
3	7	4		9	7	4		9
4			10	11			10	11

Table 3.1: DiffEq binding summary

3.1. The Centralized Management Foundation

The Centralized Management technique is based on a set of conditions that were intuitively described in section 1.2. In this section these conditions will be summarized in a theorem that will help to understand the Speculative Execution Paradigm, described later in chapter 4.

3.1.1. The Non-Speculative Execution Paradigm Theorem

If the reader remembers the example explained in section 1.2, a question about writing *Operation 6* in R2 was considered. See figure 3.1 and table 3.1. In order to write *Operation 6* result in register R2, the controller must be in state 2, the corresponding one to the cstep where *Operation 6* was scheduled. Besides, it is necessary that *Operations 1* and *2*, i.e. its predecessors, have been written before in registers R1 and R2, respectively. If other operations are considered, these conditions will be similar. For instance let's think about *Operation 11*. *Operation 11* can be written in register R4 if the controller is

in state 4 and if *Operations 7* and *9* have been written before in registers R1 and R4, respectively. Although the own state is enough for certifying that the predecessor results have been written, this separation between state and hazards will be maintained, as explained in section 1.2.

Therefore, the writing or *commit* condition can be generalized with the following theorem.

Theorem 1 (*Non-speculative Execution Paradigm Theorem* (NEPT))**.** *Let O be an operation, SC_o the associated cstep/state when O has been statically scheduled, and FU_o and R_o the FU and register where O has been bound, respectively. O can be committed iff:*

(1) The datapath controller state is SC_o.

(2) Read After Write, Write After Read, and Write After Write dependencies of O are resolved.

Proof. There are two implications. The first is,

$(1) \wedge (2) \Rightarrow$ *O can be committed*

If (1) then the inputs to FU_o will be correctly addressed and therefore the result will be properly calculated. If (2) then there are no dependencies with other operations, so writing R_o will not cause any hazard and O will be committed.

The second implication is,

O can be committed $\Rightarrow (1) \wedge (2)$

For this implication the logic rule $(A \Rightarrow B) \Leftrightarrow (\neg B \Rightarrow \neg A)$ will be used. That is,

$\neg(1) \vee \neg(2) \Rightarrow$ *O cannot be committed*

$\neg(1)$ is due to the controller state. Being in a different state from SC_o, O cannot be executed or committed. And if $\neg(2)$, writing R_o would be incorrect (in the case of RAW dependencies), or would overwrite a value that still must be read (in the case of WAR/WAW dependencies). □

Starting from this theorem, the Speculative Execution Paradigm will be defined in chapter 4, where the reader will check that these speculative conditions are not so different from the NEPT conditions.

3.2. Architecture

The Centralized Management architecture function is to generate the next state and the load and routing signals of the datapath, taking into account whether there is a misprediction or not, and satisfying the NEPT conditions. Centralized Management must comply with the NEPT because

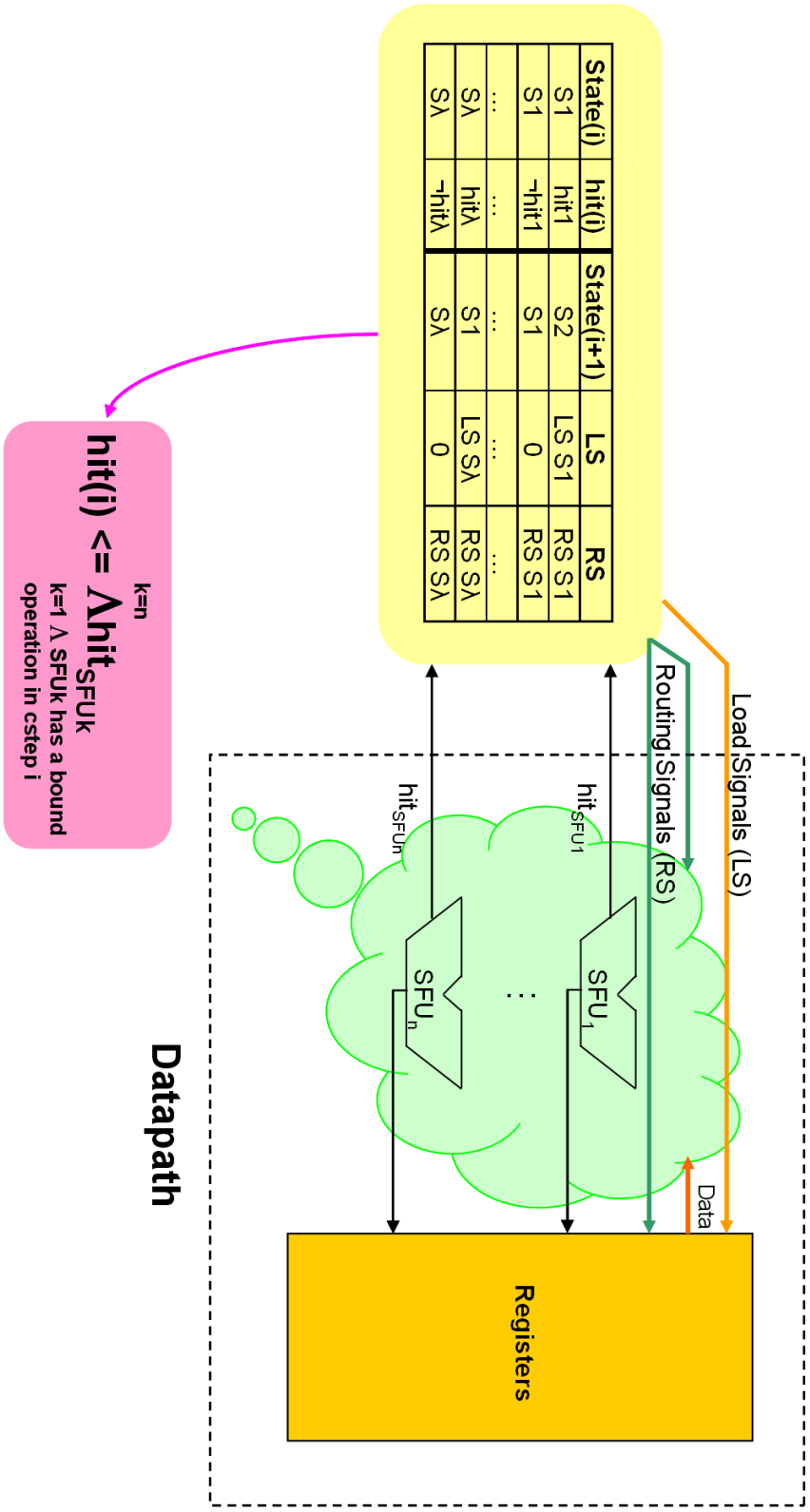


Figure 3.2: Centralized Management Architecture

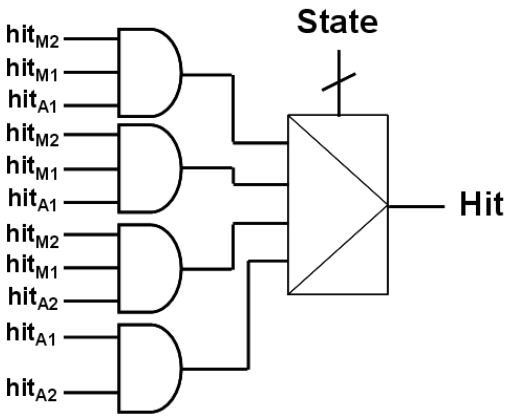


Figure 3.3: One hit signal per cstep

operations are always executed according to the initial static scheduling, although Speculative Functional Units are being used. A general overview of the Centralized Managment architecture can be seen in figure 3.2.

With Centralized Management the whole datapath is stopped everytime there is a failure. Let's consider then a generic datapath that has been scheduled in λ control-steps. The number of possible stalls in this generic datapath is equal to λ, as only one failure per cstep can be produced, independently of the number of FUs. One additional *cstep-hit signal* per cycle must be generated for notifying the controller about the failure or not. In every cstep, this *cstep-hit signal* will be the logic **AND** of all the individual hit signals proceeding from the SFUs in the datapath which execute operations in that cstep. Hence the additional hardware is quite simple because the FUs will always be synchronized.

As it can be observed the Routing Signals (RS) only depend on the controller state, i.e. they are *Moore signals*. On the other hand, the Load Signals (LS) of every state will depend on the corresponding *cstep-hit signal*, i.e. they are *Mealy signals*. Finally, the state transition will be produced only in those cases when the corresponding *cstep-hit signal* is triggered.

Figure 3.3 depicts a straightforward implementation of the aforementioned *cstep-hit signal*. It shows a hit signal per cstep (every **AND** output), and its later selection according to the corresponding state. Then, with this method there will be so many **AND** gates as the latency, i.e. λ, which will also determine the size of the multiplexer. These **AND** gates will have so many inputs as operations writing registers in every cstep.

Finally the last question that must be solved is how predictors are updated. In the Centralized Management technique this correction will be produced everytime there is a failure. As all the operations scheduled in the

same cstep than the failing operation are stopped, proceeding in this way will guarantee the synchronization of the datapath. As the correction signal is the negation of the hit signal, the writing of the predictors will be handled by the SFUs themselves, so only the failing SFUs will update their associated predictors while the others wait during the penalty time. In other words, the Centralized controller will stall those operations bound to SFUs that had guessed the carry, while providing one extra cycle to those operations bound to SFUs that had failed in the prediction.

3.3. Multicycling and chaining

The inclusion of multicycle SFUs and the chaining technique will be treated in this section. Multicycle SFUs will help to boost performance, because the penalty due to failures will be less significant than in the case of monocycle SFUs. Besides, the application of the chaining technique will be studied too.

3.3.1. Multicycle SFUs

SFUs possess two main advantages. On the one hand they reduce modules critical path and thus cycle time, but on the other hand if there is a failure the correction only happens in the last stage of the SFU. However this last feature cannot be exploited if only monocycle SFUs are considered.

If a monocycle SFU predicts the carry correctly, it will take one cycle, while if it fails it will take two. This is a 100 % penalty. Now, for example, consider a multicycle multiplier that takes 3 cycles to execute an operation. As predictors are located in the last stage, if there is a failure only the last stage will be modified, that is a minimum delay that can be modeled with one cycle, i.e. a 33.3 % penalty. Reducing FUs latency instead of cycle time will reduce the impact of the penalty cycles due to mispredictions, which will increase overall performance.

A speculative adder of size n reduces its critical path from n to $n/2$. Then the path length for correcting results is $n/2$. On the other hand, a speculative multiplier of size nxn reduces its critical path from $2n$ to $3n/2$. Note that the bits that should be corrected are the most significant $n/2$ too, so the absolute value of penalty is similar to the adders case. This fact matches with the timing analysis performed in chapter 2.

3.3.2. Chaining

Chaining is a technique that allows executing several operations which have data dependencies, i.e. RAW hazards, in the same cstep.

From an architectural point of view, chaining will affect the cstep-hit

signals generation. In every cstep all the active SFUs must be considered, so every chained SFU must be taken into account for generating the associated cstep-hit signal, as shown in figure 3.3. Nevertheless, as SFUs correction is auto-handled, if there is a failure in one of the chained SFUs, only the failing ones will be corrected.

With Centralized Management, the application of chaining does not provide better results, as it will be shown in the experiments section. In order to commit all the operations in a concrete cstep, all the chained operations must read a correct prediction. For example, consider two operations *O1* and *O2*, such that there is a RAW hazard and *O1* is a predecessor of *O2*, without loss of generality. Without chaining, *O2* will not be executed until *O1* will not have been committed. With chaining, *O2* will not be committed until *O1* will be able to be committed too. Therefore, there is no gain with the application of chaining.

3.4. Example of use

In this section the motivational example shown in figure 1.4 will be reevaluated taking into account the foundations of the Centralized Management technique, that have been explained in the previous section. Besides, an example considering multicycle SFUs will be depicted too.

3.4.1. Monocycle SFUs

As the timing analysis has already been done in the motivational example given in the introduction, this subsection will be focused on the architectural details.

See figure 3.4. Three failures happen during the execution of two iterations of the DiffEq benchmark, which will cause three penalty cycles. Let's consider for example the misprediction suffered by *Operation 5* in cycle 1. The *cstep 1-hit signal* will be the logic **AND** of the hit signals proceeding from M1, M2 and A1, because they are the active FUs in cstep 1. As the result of this condition is false, none of the operations will be written in their corresponding registers. Besides, the next cycle will be a stall one for recovering the datapath from the misprediction.

M1, M2 and A1 will evaluate their hit signals in order to update or not their corresponding predictors. As only *Operation 5* has suffered the misprediction, only A1 predictor will be updated at the end of cycle 1, so as that the correct prediction will be ready in order to perform the correction for *Operation 5* in cycle 2.

The same process will be followed with Centralized Management every time a failure happens. Finally the reader must remember that with monocycle SFUs overall performance was similar, although slightly worse, as shown

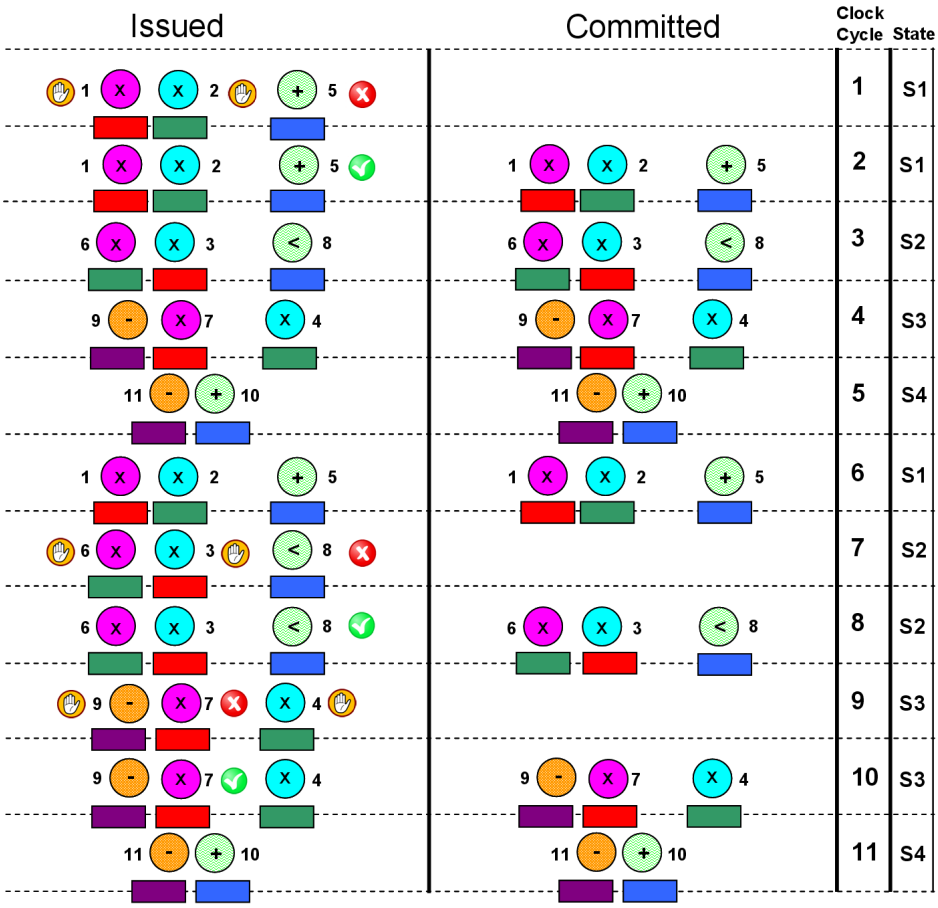


Figure 3.4: Execution of 2 iterations of the DiffEq benchmark with monocycle SFUs and Centralized Management

in section 1.3.2. This will be improved with the use of multicycle SFUs.

3.4.2. Multicycle SFUs

In order to analyze the execution flow with multicycle SFUs, first the best case scheduling must be considered. This is shown in figure 3.5. The best case scheduling is the static scheduling resulting from the assumption that there are no failures in the predictions. Then, supposing that a multiplier and an adder take 3 and 1 cycles if they hit, and one more if they do not, respectively, the best case scheduling will be performed as if the latency of both modules were always 3 and 1 cycles, respectively. Centralized Management will introduce stalls dynamically, everytime the failures happen.

On the other hand there is the worst case scheduling, i.e. the one result-

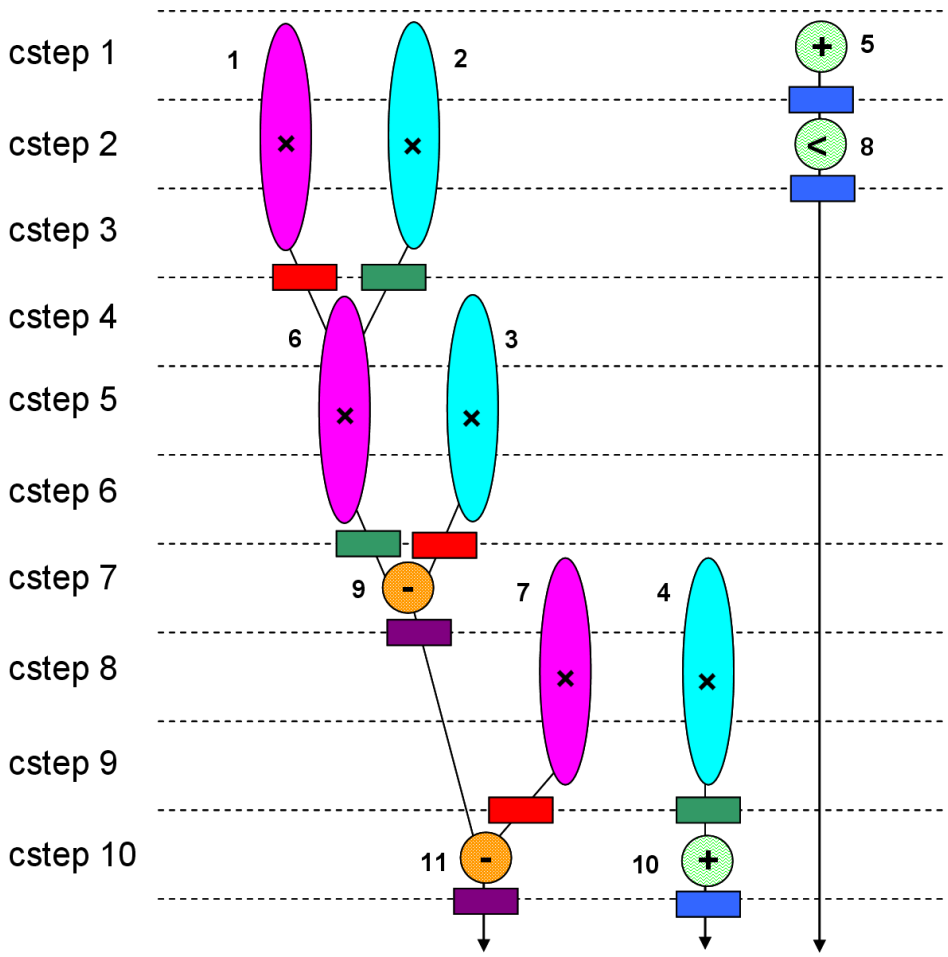


Figure 3.5: Best case DiffEq scheduling with multicycle SFUs

ing from using the failing latencies. This is the same as the case with no speculation. Then, supposing that multipliers and adders take 4 and 2 cycles in generating a result, respectively, one iteration of the DiffEq benchmark will take 14 cycles, instead of the 10 cycles that the best case scheduling lasts. This worst case scheduling will correspond to the baseline case.

Then let's examine figure 3.6, where the execution flow of the DiffEq is shown. *Operation 5* suffers a failure in cycle one, and then state S1 is stalled until the next cycle, in other words, the whole datapath is stalled. The same happens in cycles 10, 14 and 16, where 3 extra cycles will be included. Hence, if no more failures are supposed, two iterations of the algorithm will take 24 cycles, while in the baseline case it would take 2*14=28 cycles. This is a 14.3 % reduction, in comparison with the slight loss of performance that happened in the monocycle case. Note that cycle time has not been consid-

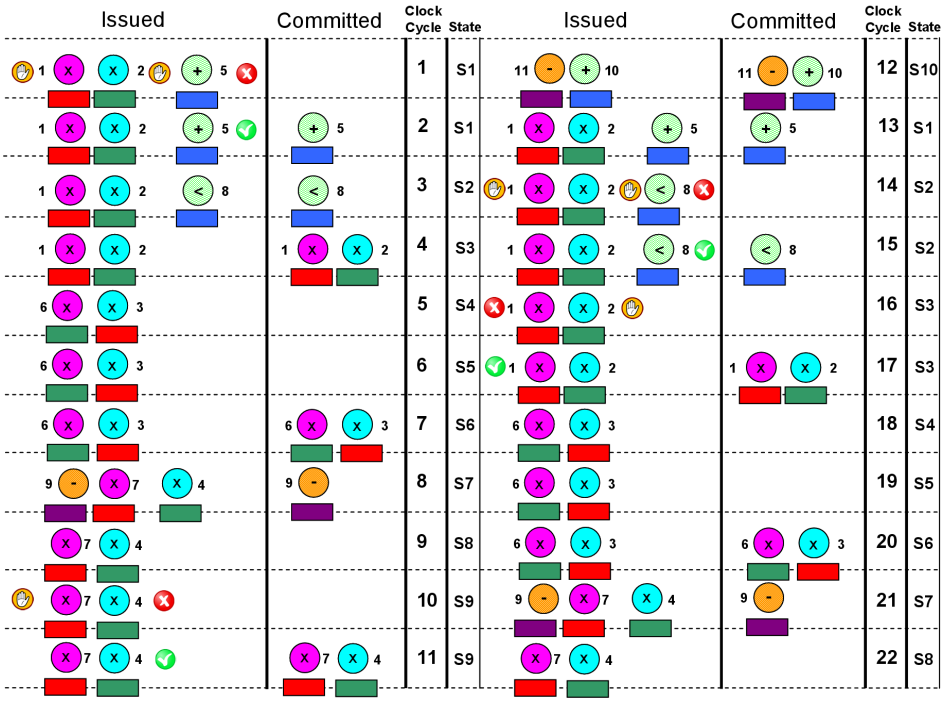


Figure 3.6: DiffEq execution flow with multicycle SFUs and Centralized Management

ered for calculating execution time in this example, because as multicycle FUs are utilized, it will be similar with and without speculation.

3.5. Experimental Results

In this section the experimental framework will be described first, and afterwards execution time and area results will be shown for comparing conventional implementations, using Ripple Carry and Carry Select adders, with another implementation using SFUs in combination with the Centralized Management technique.

3.5.1. Experimental framework

A simulator has been built for measuring performance. Simulator inputs are the DFG specifications and a bit level characterization of the most common values of the DFG inputs [BMMH07]. In this way, values can be simulated along the entire datapath and hits and mispredictions can be computed for deciding the next states of the datapath.

Given that real input patterns for the benchmarks are not available, the primary input patterns have been generated synthetically. Due to the fact

that real values are not always the same as the mean value assumed in the characterization, a *randomicity* parameter p is included in our analysis. Parameter p measures the probability of a bit to behave as it was assumed. In other words, the operands will be more similar to the pattern as p is increased, that is, a higher p value indicates higher data correlation. This value ranges from 0 to 1, but there is a symmetry between the results obtained by simulating from 0.5 to 1.0 and from 0.5 to 0. In these experiments only the p values ranging from 0.5 to 1.0 have been taken into account. In concrete, experiments have been measured for three different values of parameter p: 0.5, 0.75, 1.0.

The reader is invited to look up appendix A for more information about the data simulation along the datapaths.

The architecture is simulated at Register Transfer Level during 1,000 iterations. This could seem few time but results are similar for greater number of iterations. In order to calculate the execution time, first the average latency or *Cycles Per Iteration* (CPI) is computed, and afterwards it is multiplied by the cycle time given by *Synopsys Design Compiler*.

In order to generate the HDL implementations, a VHDL-code generator has been built too. This program maps the datapath and generates the controller. *ModelSim* is used to verify functionality of the circuits and *Synopsys Design Compiler* is used for synthesizing the designs. The target library is the *TSMC tcbn65gplustc* library, in 65 nm technology. Cycle time is measured in nanoseconds and area in μm^2.

Six benchmarks have been used in the experiments:

(1) The *Differential Equation* (DiffEq).

(2) The *Second Order Elliptic Wave Filter* (2EWF).

(3) The *Discrete Cosine Transform* (DCT).

(4) The *Inverse Discrete Cosine Transform* (IDCT).

(5) The *Lattice Filter* (Lattice).

(6) The *Least Mean Square Filter* (LMS).

The static scheduling consists in a priority list based algorithm like [WC95], while the FU-binding has been performed following a traditional resource constrained algorithm and the register binding using a left-edge algorithm [Mic94]. Experiments have been performed with 16 bits precision.

The settings of the benchmarks are shown in tables 3.2a and 3.2b. Table 3.2a depicts the number of nodes of every benchmark, the number of used adders and multipliers, and the latency of the circuit with no SFUs when using both monocycle and multicycle FUs. Table 3.2b shows the latency for

Benchmark	Nodes	+	*	λ_{mono}	λ_{multi}
DiffEq	11	2	2	4	14
2EWF	34	3	2	14	32
DCT	40	3	3	10	30
IDCT	40	3	3	10	28
Lattice	13	2	1	9	26
LMS	17	2	2	9	26

(a) Number of nodes, adders, multipliers and circuit latency

FU	λ	λ_{hit}
+	2	1
*	4	3

(b) Multicycle FU latencies

Table 3.2: Experiment settings

multicycle FU when using non-speculative and speculative FUs, respectively. Note that the SFUs latency that appears inn the table is the one supposing that the prediction is a hit. When there is a misprediction, the latency will be the same as the non-speculative case. In the case of Carry Select Adders, the latency will be the same as in the speculative case.

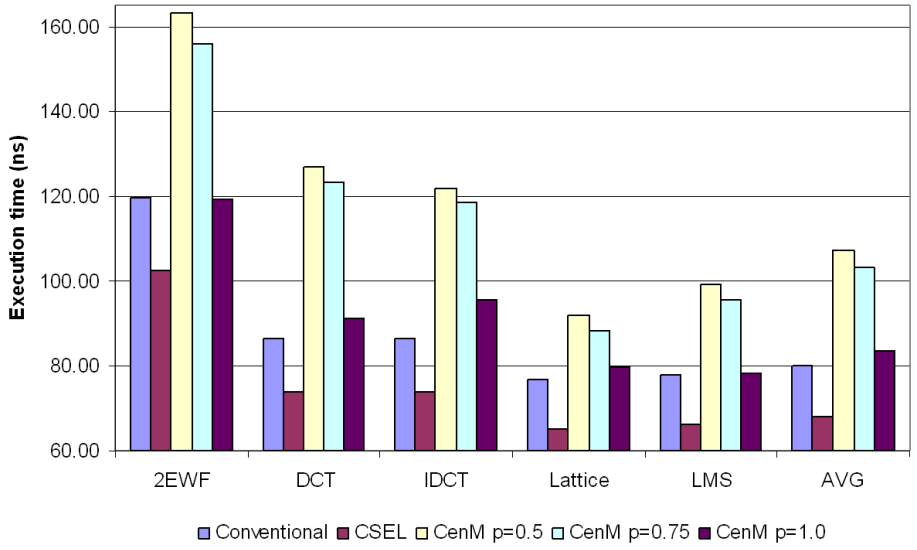
Ripple Carry Adders and Baugh-Wooley Multipliers are the non-speculative FUs considered in the baseline case, while the Predictive Adders and the Predictive Multipliers explained in sections 2.1.6 and 2.2.3, respectively, are the Speculative Functional Units that will be deployed in the speculative case.

Finally, note that in the latency and execution time figures, the results of the DiffEq benchmark are not shown because they degrade the scale of the charts. In spite of this, they also contribute to the average results.

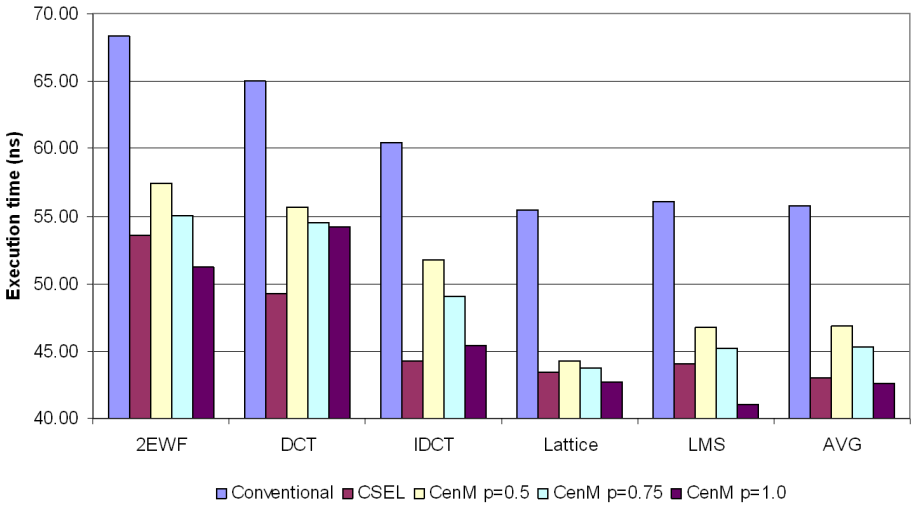
3.5.2. Synthesis results

The aforementioned six benchmarks have been simulated in order to obtain execution time results. Simulations have been performed with non-speculative Ripple Carry Adders and Baugh-Wooley Multipliers with a RCA in the last stage, which is labeled as conventional case, and with Speculative Functional Units under Centralized Management. Besides, a comparison with a conventional implementation using Carry Select Adders and Baugh-Wooley Multipliers with a CSEL in the last stage is included.

First a study with monocycle FUs has been performed. Execution time



(a) Execution time with monocycle SFUs



(b) Execution time with multicycle SFUs and chaining

Figure 3.7: Execution time with Centralized Management

results are shown in figure 3.7a and the cycle time ones in table 3.3. As it can be observed the Centralized Management results are worse than using a conventional implementation. Although cycle time is lower in the case of Centralized Management, average latency penalizes too much overall performance, because of the stalls introduced in the datapath every time there is a failure.

Benchmark	Conventional	CSEL	CenM
DiffEq	8.35	7.09	6.74
2EWF	8.56	7.31	7.46
DCT	8.67	7.38	7.31
IDCT	8.64	7.38	6.92
Lattice	8.54	7.23	6.93
LMS	8.63	7.34	7.11
AVG	8.56	7.29	7.08

Table 3.3: Cycle time summary with monocycle FUs

On the other hand, execution time decreases while increasing parameter p. This is an obvious consequence, because of the correlation-based nature of predictors. Just remember that p measures the probability of a bit to behave as it was assumed. In other words, as p is increased the primary inputs to the circuit will become more similar, so they will produce more similar carries, i.e. the hit rates will increase. Note that this effect will happen in every experiment which considers parameter p. Hence it can be concluded that Centralized Management is worse than the conventional implementation between 34.3 % (p=0.5) and 4.5 % (p=1.0). Evidently, CSEL implementation is much better (22.8 % execution time less than the best case with Centralized Management, i.e. p=1).

Now let's examine cycle time results with monocycle FUs. The first conclusion that can be obtained is the cycle time reduction in both the CSEL and the Centralized Management implementations, with respect to the conventional one. In addition, it is interesting to note that the CSEL implementation cycle time is slightly greater than the Centralized Management one. This is due to the extra multiplexer that is included in the CSEL design.

The second consequence that can be extracted from table 3.3 is that the theoretical gain of the Predictive Multipliers is not reached. As 16-bits primary inputs are being used, and homogeneous datapaths considered, the maximum delay during a cycle can be estimated as the 16x16 multiplier delay. According to results from table 2.5, cycle time should be reduced around 21 %, which is not the case. Actually, on average, cycle time is reduced 17.3 % with Centralized Management. The slight loss of cycle time reduction is due to the additional control imposed by the Centralized Management itself. Nevertheless, as controller is developed independently of data width, this additional delay will be similar in absolute terms, although it will become less significant in relative terms when increasing data width. In other words, with higher data widths cycle time will be closer to the theoretical 25 % reduction explained in section 2.4.

3.5.2.1. Multicycle FUs and chaining impact

The second study has been performed with multicycle FUs and considering the chaining technique too. These results are shown in figure 3.7b. Note that in this second study the baseline case has considered multicycle FUs and chaining too. Cycle time is estimated by dividing the monocycle clock period by the greatest FU latency in the design. As homogeneous datapaths (those with uniform word length) are being considered, the module with the greatest latency will be the multiplier, whose latency is 4 and 3 cycles for the non-speculative and speculative case, respectively. For example, the estimated cycle time with multicycle FUs of the DiffEq benchmark will be 8.35/4 ns and 6.74/3 ns for the conventional implementation and the Centralized Management one, respectively.

The use of multicycle SFUs with Centralized Management reduces the execution time of the conventional implementation. This is due to the smaller weight of the penalty cycles. It is possible to diminish execution time in 24.2 % on average, and in 28 % in the best case. These results are close to those obtained with Carry Select Adders. However Centralized Management only equals performance when parameter p is very close to 1.0, which is fairly impractical. Primary inputs do not behave always as the average pattern. Therefore, supposing a certain degree of input correlation, the more practical results should be closer to those obtained with p=0.75. Hence, with p=0.75, the use of SFUs with Centralized Management diminish execution time in 18.7 % on average, with respect to a conventional implementation with RCAs, but it is still 5.3 % worse with respect to the CSEL implementation.

Note that the impact of chaining over Centralized Management does not produce any significant performance difference. Execution times are practically equal when considering multicycle FUs and when considering them jointly with the chaining technique.

3.5.2.2. Area penalty analysis

Finally, area results are shown in figure 3.8. Every set of bars depicts the area of the benchmarks with monocycle non-speculative FUs (Conventional), a conventional implementation with Carry Select Adders (CSEL), with monocycle SFUs and Centralized Management (CenM), with multicycle non-speculative FUs (Conv+Mult), a Conv+Mult implementation but with Carry Select Adders (CSEL+Mult), and with multicycle SFUs and Centralized Management (CenM+Mult).

Note that chaining impact over area is negligible, because it only implies a change in the initial scheduling and/or binding, i.e. changing the controller signals that are active every cstep. Hence, the controller will have a similar complexity to the original one. On the other hand, the chaining technique can collaborate to reduce the number of registers as well as the circuit latency.

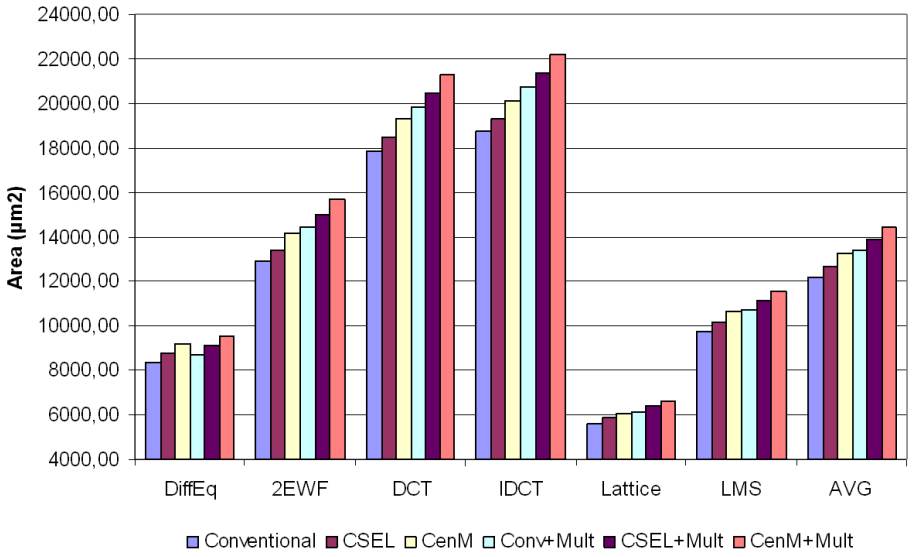


Figure 3.8: Area results with monocycle or multicycle SFUs, and with or without Centralized Management

But if the latency and/or resource constraints only allow to chain a few operations at the most, the number of registers will remain roughly the same. As in the experiments a hard latency constraint and a fixed number of FUs are being used, there are few free resources in every cstep to chain operations. Thus, the area of the implementation with chaining will be similar to that with no chaining.

As it can be observed there is not much difference between monocycle and multicycle experiments, due to most of the benchmarks area proceeds from the datapath. Centralized Management area penalty is 8.8 % and 7.8 % with monocycle and multicycle SFUs, respectively. As in the case of delay, these relative ciphers will decrease their significance as data width increases, because controllers and predictors will remain the same independently of data size.

Finally, it should be noted that CenM occupies 4.4 % and 4 % more area than the CSEL case, with monocycle and multicycle FUs, respectively. This makes CenM not suitable, specially in the case of monocycle FUs, since its performance is clearly worse than the CSEL one. However, area penalty deserves a deeper study. See figure 3.9. CenM area penalty is due to the controller, while the CSEL penalty is due to the data width. In other words, the larger the adder, the larger the penalty. The reason is clear: half of the adder will be replicated and one multiplexer will be included too in every Carry Select Adder of the design. Therefore, as data width increases, CSEL

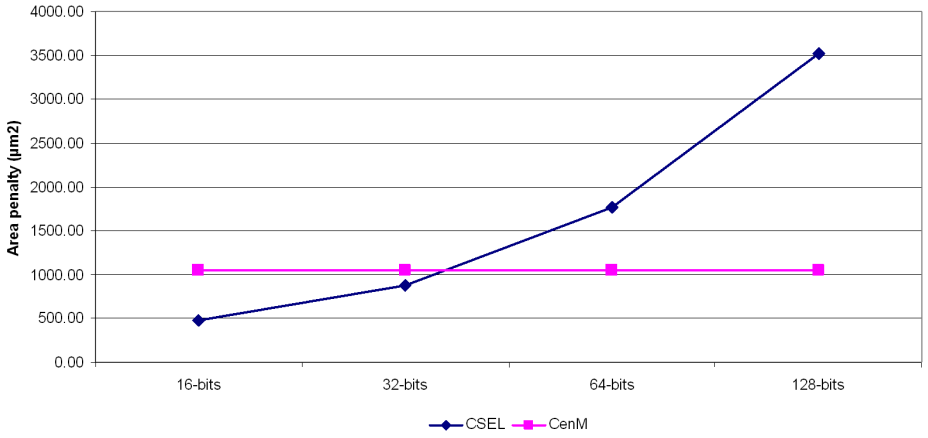


Figure 3.9: Area penalty with respect to data width

area penalty will increase too, while in the case of CenM it will remain constant. Finally note that area overhead is the same with monocycle and with multicycle FUs, because the conventional, the CSEL and the CenM implementations have the same Finite State Machine (in spite of the fact that CenM includes some additional control for introducing stalls in the datapath).

3.5.2.3. Using logarithmic FUs

In order to test the behavior of Centralized Management with different and faster SFUs, logarithmic designs such as the ones explained in subsection 2.1.5 have been studied. Multicycle SFUs have been considered in this experiment. Moreover, the reader must take into account the time analysis performed in subsection 2.3 and that the baseline case utilizes non-speculative logarithmic FUs.

As these designs come from the works described in [VBI08, Cil09], and have been synthesized with a different target technology, neither cycle time or area results are applicable. Hence, only a latency comparison can be given. But as the delay will be reduced around 1.5 times in comparison with a conventional logarithmic adder, according to [VBI08], it can be assumed that a reduction in latency will be translated into a further reduction in overall execution time. Note that chaining has not been applied in this experiment. If chaining were applied with logarithmic FUs, cycle time would be increased both in the speculative and the non-speculative case, but as the quantity is uncertain because of the aforementioned reasons, chaining has not been considered. This does not happen with linear structures such as the RCA, because the chaining of a few additions only increases the critical path in a

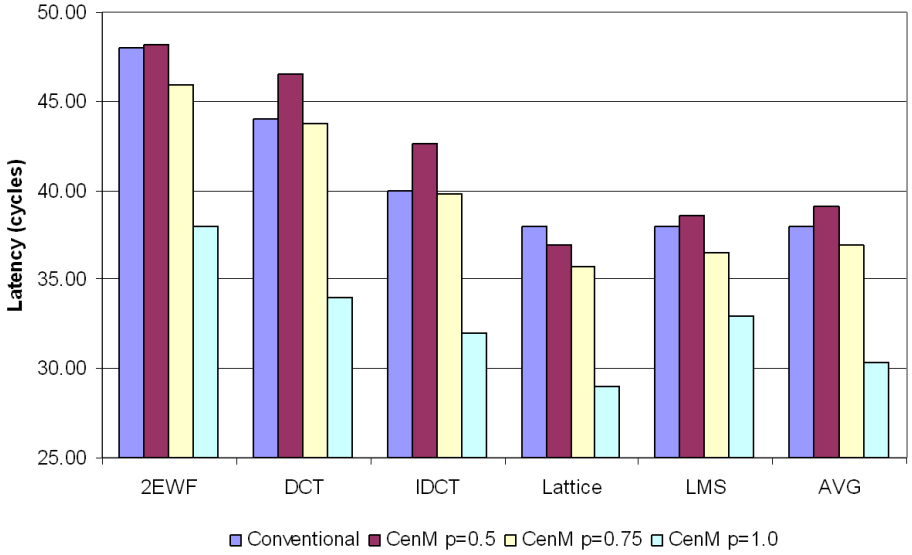


Figure 3.10: Latency with logarithmic FUs

few Full Adders. In fact, this increase is negligible in the previous experiments because of the stringent resources restrictions.

Figure 3.10 shows the latency results. As it is observed, CenM diminishes latency significantly only with very high correlation values, achieving 20.2 % reduction with p=1.

Chapter 4

Distributed Management of SFUs in HLS

The important thing is not to stop questioning

Albert Einstein

Is it possible to go beyond Centralized Management? Stopping all the operations every time there is a failure is really necessary ? This chapter will address these questions and will provide the foundations of an improved management in order to handle Speculative Functional Units.

The main idea behind *Distributed Management* (DisM) is to allow operations to be executed by some SFUs, independently of misprediction events in other SFUs [BMM+10, BMM+11]. It is inspired in the dynamic scheduling of processors proposed by Tomasulo et al. [AST67, Tom67, HP07]. This algorithm allowed the execution of operations that were ready, but stalled because of a previous non-executed operation. Thus, the same principle can be applied when considering SFUs. If there is a misprediction, but there are operations that can be executed, why must the datapath be stalled completely?

Nevertheless, in High-Level Synthesis there are some special issues to be considered. The first problem is the maintenance of controller state. If there is a failure in a SFU and not all SFUs are stopped, there will be some SFUs working in different states at a given instance. Therefore, one dedicated controller per SFU is required. Distributed Management proposes to split the global controller into several smaller controllers embedded in the datapath. Besides, as every operation can happen independently from others, one state value per operation will be necessary.

Similar to processors, there will appear *Read After Write* (RAW), *Write After Read* (WAR) and *Write After Write* (WAW) dependencies [HP07]. In

processors, shelving techniques, reservation stations, and reorder buffers are used to ensure the correct behaviour of the program. However in HLS these techniques cannot be afforded because of their area and power overhead. For resolving these dependencies, Distributed Management will instead exploit the program knowledge, i.e., the Dataflow Graph.

In the following, a set of definitions, theorems, and lemmas that will be used for generating hazard-free datapaths will be presented, as well as the proposed new architecture for the seamless deployment of SFUs in HLS.

4.1. The Distributed Management Foundation

In this section the theoretical foundation for designing a datapath controller that can utilize SFUs with the Distributed Management mechanism is presented. This will derive in the construction of a canonic architecture that will be the basis for applying this new technique automatically to every benchmark.

4.1.1. The Speculative Execution Paradigm Theorem

This subsection will introduce some concepts and, similarly to the non-speculative case, it will postulate a theorem in order to summarize all the conditions that every operation must satisfy when working with SFUs and thus keeping datapaths in a correct state.

Definition 1. *An operation is executing (or is in execution) iff it is being computed in its corresponding FU.*

Definition 2. *An operation commits iff it writes in its corresponding register and produces the corresponding state transition in the controller associated to the FU that has executed it.*

Definition 3. *An operation is non-committable in a given cycle iff it cannot be written in its corresponding register.*

Definition 4. *Every resource imposes an order given by the sequence of operations bound to it.*

Before deriving the conditions that operations must verify in order to be executed and written correctly in a speculative paradigm, it is mandatory to review firstly what conditions are verified in the non-speculative paradigm. These were summarized in the Non-speculative Execution Paradigm Theorem, which was postulated in section 3.1.1.

Consider the DiffEq example given by figure 4.1 and whose binding summary is depicted in table 4.1. According to the NEPT, *Operation 6* can only be executed in M1 and written in R2 if the datapath controller is in cstep 2

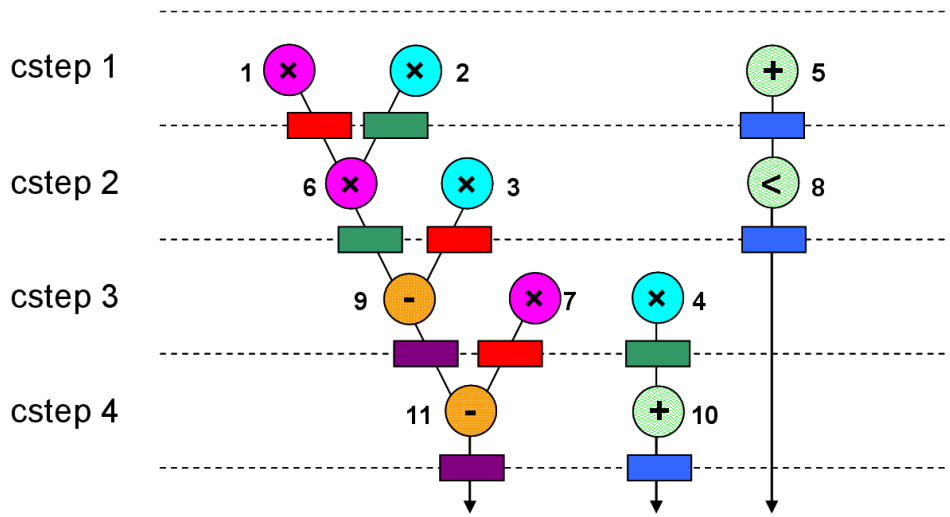


Figure 4.1: DiffEq scheduling and FU and register binding

cstep	M1	M2	A1	A2	R1	R2	R3	R4
1	1	2	5		1	2	5	
2	6	3	8		3	6	8	
3	7	4		9	7	4		9
4			10	11			10	11

Table 4.1: DiffEq binding summary

and if *Operations 1* and *2* have been written in R1 and R2 respectively during the previous cstep. In a conventional implementation, or using Centralized Management, these conditions are guaranteed by the static scheduling and binding. But this is not possible if different operations are allowed to be at different states at the same time. The hazards condition should be checked dynamically to certify the correctness of the circuit. However, it would not be possible to check the state condition because the state is the same for the whole datapath. Hence, as every FU can only execute one operation at a given cstep, one state variable per FU is required.

Nevertheless, in spite of the abovementioned facts, by introducing the possibility of mispredictions only one new variable is being considered, so commit conditions in the speculative execution paradigm should be very similar to the NEPT conditions. However, the fact of using several FU-controllers must be taken into account, i.e. there will be several states. This is captured in the following theorem,

Theorem 2 (*Speculative Execution Paradigm Theorem* (SEPT))**.** *Let O be an operation in execution. Let SFU_o and R_o be the SFU and register where*

it has been bound, respectively. Let St_{SFUo} be the local state of SFU_o. Let S_o be the associated state to O. Then, O commits iff

(1) The carry prediction of SFU_o is a hit

(2) St_{SFUo}=S_o

(3) Read After Write, Write After Read and Write After Write dependencies of O are resolved

Proof. There are two implications. The first is

$(1) \wedge (2) \wedge (3) \Rightarrow O$ *commits*

Condition (2) means that the inputs to SFU_o will be correctly routed. If (1), then the result produced by SFU_o is correct, because there is no failure in the prediction. Finally if (3), then there are no dependencies with other operations, so writing R_o will not cause any hazard and O will be committed. Therefore, the conditions are the same as if there were no dynamic scheduling and O will be committed, i.e. written in R_o.

The second implication is

O *commits* $\Rightarrow (1) \wedge (2) \wedge (3)$

For this implication, the logic rule $(A \Rightarrow B) \Leftrightarrow (\neg B \Rightarrow \neg A)$ will be used. That is,

$\neg(1) \vee \neg(2) \vee \neg(3) \Rightarrow O$ *cannot commit*

which is trivially true. If $\neg(1)$ then the result is not correct, if $\neg(2)$ then St_{SFUo} is different from S_o, so it is not the turn of O to be executed, and if $\neg(3)$ then there is a dependency that breaks the correctness of the program if the result is written in R_o.

□

In the example of figure 4.1 consider *Operation 6*. As it has been said in the introduction of this chapter, there is a state value per operation. Thereby, the state value associated to every operation will be given by the own operation identifier. Thus, *Operation 6* will correspond with the state value S6, *Operation 1* with S1, and so on. Hence, *Operation 6* will be committed if the local state controller is in the correct state where *Operation 6* was scheduled, i.e. S6, and if *Operations 1* and *2* have been committed previously, i.e. RAW and WAW hazards have been resolved, and if there is a hit in

the prediction of the SFU which is executing *Operation 6*. Note that with the speculative execution paradigm a WAW hazard has appeared between *Operations 2* and *6*, which did not exist with the non-speculative paradigm.

Therefore speculative conditions are somehow similar to the non-speculative ones, but with two important differences: the hit condition, and the use of several local states instead of only one global state.

Note that the dynamically checking conditions problem did not exist with the NEPT, because hazards were resolved statically by construction of the scheduling and binding. Moreover, note that this problem neither existed with the Centralized Management technique, because the hazards were solved statically by construction of the scheduling and binding too (condition (3)), and the global state was always synchronized with the cstep, due to the global stalls provoked by mispredictions (condition (2)). Hence, the only condition to be checked, in the case of Centralized Management, is condition (1). In fact, this is what has been explained in section 3.2 with figure 3.3.

The problem that arises from the SEPT is the verification of Condition (3). Condition (1) is easily verifiable; it suffices to check the hit signal of the SFU where the operation under consideration has been bound. Condition (2) can be verified just by checking the value of the local state of the corresponding SFU. The question is how to verify that every hazard is resolved in a concrete cycle, i.e. in execution time, while keeping a minimal hardware overhead.

The proposed solution consists in checking the FUs local states where operations that cause dependencies have been bound. For example, consider the RAW and WAW hazards that exist between *Operations 6* and *2*. The question that must be answered is: *"Has Operation 6 solved its dependencies with 2?"* Hence we must check the state of M2, where *Operation 2* must be executed. *Operation 2* has been scheduled in state S2, so if the state of M2, i.e. St_{M2}, is S3 or S4 we can conclude that *Operation 2* has been committed and thereby the hazards have been resolved. This is the basic idea for checking dependencies.

In order to decide whether or not an operation has resolved its dependencies, first how the system is supposed to work must be defined. Therefore several design rules will be presented to establish the basis for the correctness of the generated circuits.

4.1.2. Design Rules for generating hazard-free datapaths

Design Rules are conditions, imposed by designers, that will guide the design process. Then in this subsection a set of rules is first presented to certify that the final generated circuits will be free of hazards.

Afterwards several lemmas will be derived to summarize the specific con-

ditions that operations must verify in order to be committed without violating RAW, WAR, WAW or structural dependencies.

Design Rule 1 (DR1). *Let O_e be an operation in execution and R_e the register where it will be written. We need to ensure that the previous operation bound to the same FU as O_e, has been committed.*

Design Rule 2 (DR2). *Committed Operations Rule. Let O_c be a committed operation, and R_c the register where O_c has been written. We ensure that:*

(1) Its immediate predecessors have been committed (RAW)

(2) The previous operation bound to the same FU as O_c, has been committed (structural dependencies)

(3) The previous operation written in R_c, has been committed (WAW)

(4) If O_r is an operation that is reading the previous value in R_c, then O_r has been committed (WAR)

Design Rule 3 (DR3). *Let O_e be an operation in execution and R_e the register where it will be written. We need to ensure that:*

(1) Its successors cannot be committed (RAW)

(2) The next operation bound to the same operator as O_e, cannot be committed (structural dependencies)

(3) The next operation that will write in R_e, cannot be committed (WAW)

Design Rule 4 (DR4). *Non-Committable Operations Rule. Let O_{nc} be a non-committable operation, and R_{nc} the register where O_{nc} will be written. We must ensure that,*

(1) Its successors cannot be committed (RAW)

(2) The next operation bound to the same operator as O_{nc}, cannot be committed (structural dependencies)

(3) The next operation that will write in R_{nc}, cannot be committed (WAW)

(4) If O_w is an operation that will write in one register that O_{nc} is reading, then O_w cannot be committed (WAR)

Lemma 2 (Correctness Lemma). *A datapath designed following rules DR1-DR4 is correct.*

Proof. In order to demonstrate this lemma the logic rule $(A \Rightarrow B) \Leftrightarrow (\neg B \Rightarrow \neg A)$ will be used. That is,

$$A \text{ circuit is incorrect} \Rightarrow \neg(DR1) \vee \neg(DR2) \vee \neg(DR3) \vee \neg(DR4)$$

A circuit is incorrect if a hazard happens. Then,

(1) If a RAW hazard happens, then DR2.1 and DR3.1 or DR4.1 are violated.

(2) If a WAR hazard happens, then DR2.4 and DR4.4 are violated.

(3) If a WAW hazard happens, then DR2.3 and DR3.3 or DR4.4 are violated.

(4) In the case of a structural hazard there are two possibilities: the hazard is due to a conflict caused by FU sharing or by register sharing when writing an operation. The case of registers is the same as WAW hazards. On the other hand if there is a structural hazard with a FU, then either DR1, DR2.2 or DR3.2 or DR4.2 are violated.

□

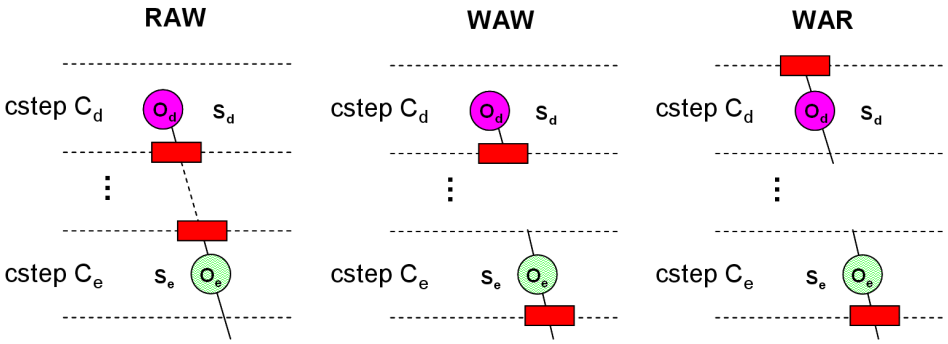
Note that DR2.2, DR3.2 and DR4.2 are not necessary for keeping the correct state of the datapath (i.e. if a later operation is ready it could be committed), but they are introduced in order to impose the same ordering as determined by the static scheduling and binding. Besides, being consistent with both the static scheduling and binding simplifies the controller design.

Definition 5. *Let S_i and S_j be two state values of the same FU controller, associated with operations O_i and O_j, respectively. Then, S_i ">" S_j iff S_i happens after S_j*

Lemma 3 (RAW/WAW Hazards Lemma)**.** *Let O_e be an operation that has not been committed yet. Let O_e have a RAW/WAW dependency with operation O_d. Let FU_e and FU_d be the functional units where O_e and O_d have been respectively bound. Let S_e and S_d be the associated states of O_e and O_d respectively when they are executed in FU_e and FU_d. Then, the hazard between O_e and O_d is resolved iff the controller of FU_d is in a state that is later than S_d, i.e. St_{FUd} ">" S_d.*

Proof. This lemma has two implications. The first implication is:

$$RAW/WAW \text{ hazard resolved} \Rightarrow St_{FUd} \text{ ">" } S_d$$



(a) RAW Hazard (b) WAW Hazard (c) WAR Hazard

Figure 4.2: Hazards scheme

If the RAW/WAW hazard (see figures 4.2a and 4.2b) has been resolved, O_d has been committed surely, because due to the static binding O_d and O_e have been statically scheduled in different csteps, so St_{FUd}, is later than S_d.

The second implication is

$$St_{FUd} \text{ ">" } S_d \Rightarrow \textit{RAW/WAW hazard resolved}$$

If St_{FUd} ">" S_d, then O_d has been committed, and therefore the RAW/ WAW hazard has been resolved. □

Lemma 4 (WAR Hazards Lemma)**.** *Let O_e be an operation that has not been committed yet. Let O_e have a WAR dependency with operation O_d. Let FU_e and FU_d be the functional units where O_e and O_d are executed, respectively. Let S_e and S_d be the associated states to O_e and O_d when they are executed in FU_e and FU_d. Then, the hazard between O_e and O_d is resolved iff the controller of FU_d is in a state later than S_d, i.e. St_{FUd} ">" S_d, or if it is in the state S_d, i.e. $St_{FUd} = S_d$, and O_d verifies the SEPT conditions (SEPT(O_d)) for being committed.*

Proof. The first implication is

$$\textit{WAR hazard resolved} \Rightarrow St_{FUd} \text{ ">" } S_d \vee \{St_{FUd}{=}S_d \wedge \text{SEPT}(O_d)\}$$

If the WAR hazard (see figure 4.2c) has been resolved, then O_d has been committed (St_{FUd} ">" S_d) or it is being executed in the local state where it was statically scheduled. In this second case if the WAR hazard is resolved, the other conditions for committing O_d (derived from SEPT) must be also

satisfied.

The second implication is

$$St_{FUd} \text{ ">" } S_d \vee \{St_{FUd}{=}S_d \wedge \text{SEPT}(O_d)\} \Rightarrow \textit{WAR hazard resolved}$$

St_{FUd} ">" S_d case follows the same demonstration as for RAW/WAR hazards. Therefore, only the $St_{FUd}{=}S_d$ case will be considered. If $St_{FUd}{=}S_d$ and O_d satisfies the SEPT conditions, then O_d is ready for being committed and therefore, O_e can write the register that O_d is reading, which was causing the WAR hazard. □

In these *Hazards Lemmas* the reader should notice that $FU_d \neq FU_e$, without loss of generality. If they were the same FU, the own state of the FU controller would indicate if the hazard is solved or not.

Lemma 5 (Structural dependencies lemma)**.** *There will not be two operations accessing to the same resource in the same cycle.*

Proof. This is satisfied by construction, thanks to DR2.2, DR3.2 and DR4.2 (structural FU dependencies) and thanks to DR2.3, DR3.3 and DR4.3 (structural register dependencies). □

4.1.2.1. Compatible States

As stated in the aforementioned lemmas, in order to resolve hazards it is enough to check the local states. In the given example it would be enough to check that St_{M2}=S3 or St_{M2}=S4. This works for a small benchmark such as DiffEq, but in general, checking the state for every possibility would introduce an area/power overhead that should be avoided. This can quickly become highly complex. Besides, consider that this condition is only for one dependency; if an operation had several dependencies with operations bound to different FUs, the final hazard-free condition would be even more complex.

A solution is proposed to limit the number of possible states of the local FU controller that need to be checked. It must be observed that not every state can happen, provided that a concrete operation is being executed. Those states that can happen are called *Compatible States*. Note that as there is a bijection between states and operations, i.e. one state per operation and vice versa, talking about *Compatible Operations* will be the same as *Compatible States*. The pseudocode of the algorithm for identifying *Compatible States* is given in algorithm 4.1.

Let's suppose that the operation O_e is in execution and that there is a data dependency with O_d, which forces to commit O_d before O_e. The idea is to determine those states from the FU controller that causes the dependency

(FU_d) that have been committed surely before O_e (*getCommittedStates(*St_e*)*), and those states that certainly cannot occur, provided that O_e is in execution (*getNonCommittableStates(*St_e*)*). Then, it is possible to find what states of FU_d can happen while O_e is being executed. These states are those neither belonging to the *Committed* nor the *Non-Committable* states. Besides, to this list we must add the first state of FU_d that belongs to the *Non-Committable* list, because one operation can be executed, i.e. it has reached the state under question, but not committed.

Note that *Compatible States* can be separated from O_e by at most one iteration. These states range between some committed states and some non-committable ones. If one state is committed in iteration i, then it has been committed in the previous iterations. Similarly, if one state is non-committable in iteration i, then it will be non-committable in the following iterations. This means that the time-window of states separating committed and non-committable states spans at most one iteration, i.e. from one state until the same state in the previous/following iteration. Hence, committed states can be at the iteration prior to or in the same iteration as O_e, and non-committable states can be in the same or in the following iteration with respect to O_e. Therefore, a *Compatible State* can be in the previous, in the same, or in the following iteration with respect to O_e. For more information, the reader is invited to investigate the details of this algorithm in appendix B.

For example, consider *Operation 6* and its RAW dependency with *Operation 2*. *Operation 2* is executed in M2, so St_{M2} ">" S2 shoud be checked, i.e. St_{M2}=S3 or St_{M2}=S4. However the M2 controller cannot be in state S4, if *Operation 6* is being executed. As *Operation 6* is just being executed, we cannot infer that *Operation 3* will be committed in the same cycle, because of the WAR hazard between *Operations 6* and *3*, and therefore the M2 controller cannot be in S4. Hence, checking St_{M2}=S3 is enough to guarantee that *Operation 2* has been committed.

Besides, note that there could be some hazards between the last operations of iteration i and the first ones of iteration *(i+1)*. For instance consider *Operation 1* and its WAR hazard with *Operation 11*, provoked by the use

Algorithm 4.1 getCompatibleStates(St_e,St_d)

1: $Comm \leftarrow getCommittedStates(St_e)$
2: $NComm \leftarrow getNonCommittableStates(St_e)$
3: $CS1 \leftarrow \{St : St \in \{St_{FUd}\} \wedge St \neq St_d \wedge St \notin Comm\}$
4: $CS2 \leftarrow \{St : St \in \{St_{FUd}\} \wedge St \neq St_d \wedge St \notin NComm\}$
5: $CS \leftarrow CS1 \bigcup CS2$
6: $CS \leftarrow CS \bigcup \{First\ St \in \{St_{FUd}\} : St \neq St_d\ \wedge St \in NComm\}$
7: **return** CS

of register R1. We must check that St_{A2} ">" S11, which is translated into St_{A2}=S9 (of the following iteration). Interiteration hazards will introduce some additional problems, which will be discussed later in section 4.2.3.

4.2. Architecture

Based on the theory developed in the previous section, the Distributed Management canonic architecture will be described. In addition it will be applied to the concrete DiffEq example in order to fully understand it.

The Distributed Management architecture is depicted in figure 4.3. Its main activity is to dynamically verify the SEPT for each operation. Every SFU is associated with one local state controller and one T-flipflop, whose use will be explained later. Every block communicates with the global *Commit Signals Logic Unit* (CSLU) by means of the SFU hit signal (hit_{SFU}), the SFU state (St_{SFU}), and the *SFU-T* (content of the T-flipflop) value.

The CSLU generates the *commit* or *enable* signals that will indicate both when a SFU controller can make the transition to the next state and when the corresponding register can be written. Besides, the CSLU generates the signals that will control the rest of the datapath, namely: multiplexers, registers, etc. The CSLU is composed of several *Enable Generation Units* (EGUs), which are the necessary logic for implementing the corresponding SFU enable signal. As everytime only one operation can be executed by a SFU, there will be only one enable per SFU and thus, one EGU per SFU.

In the DiffEq example, let's consider the instance when the controller is making the decision on committing *Operation 6*. Applying the first two conditions of the SEPT, *Operation 6* will be committed if hit_{M1}='1', and St_{M1}=S6, which corresponds with the static scheduling cstep for *Operation 6*. Conditions (1) and (2) are easy to verify. Condition (1) is verified using the hit signal output from the SFU predictors. As for the second condition, the current state is an internal signal generated by every SFU controller. In order to verify the last condition, the controller needs to know the list of *Compatible States* of operations with RAW, WAR, or WAW dependencies on the operation under question. This list is generated according to the aforementioned *Design Rules* by pre-processing the DFG statically. The necessary conditions for committing every operation are then implemented through a combinational logic network within the CSLU. Therefore, this logic is automatically derived from the information given by the scheduling and binding. As it will be shown in the experiments, it introduces a negligible area overhead.

According to DR1 and DR2, the *Committed States* list of M2, while executing *Operation 6* in M1, is composed by {S3, S4}. In this case, S3 and S4 belong to the previous iteration to that of *Operation 6*. According to DR3 and DR4, the *Non-Committable* states are {S3, S4, S2}. Note that S3 and

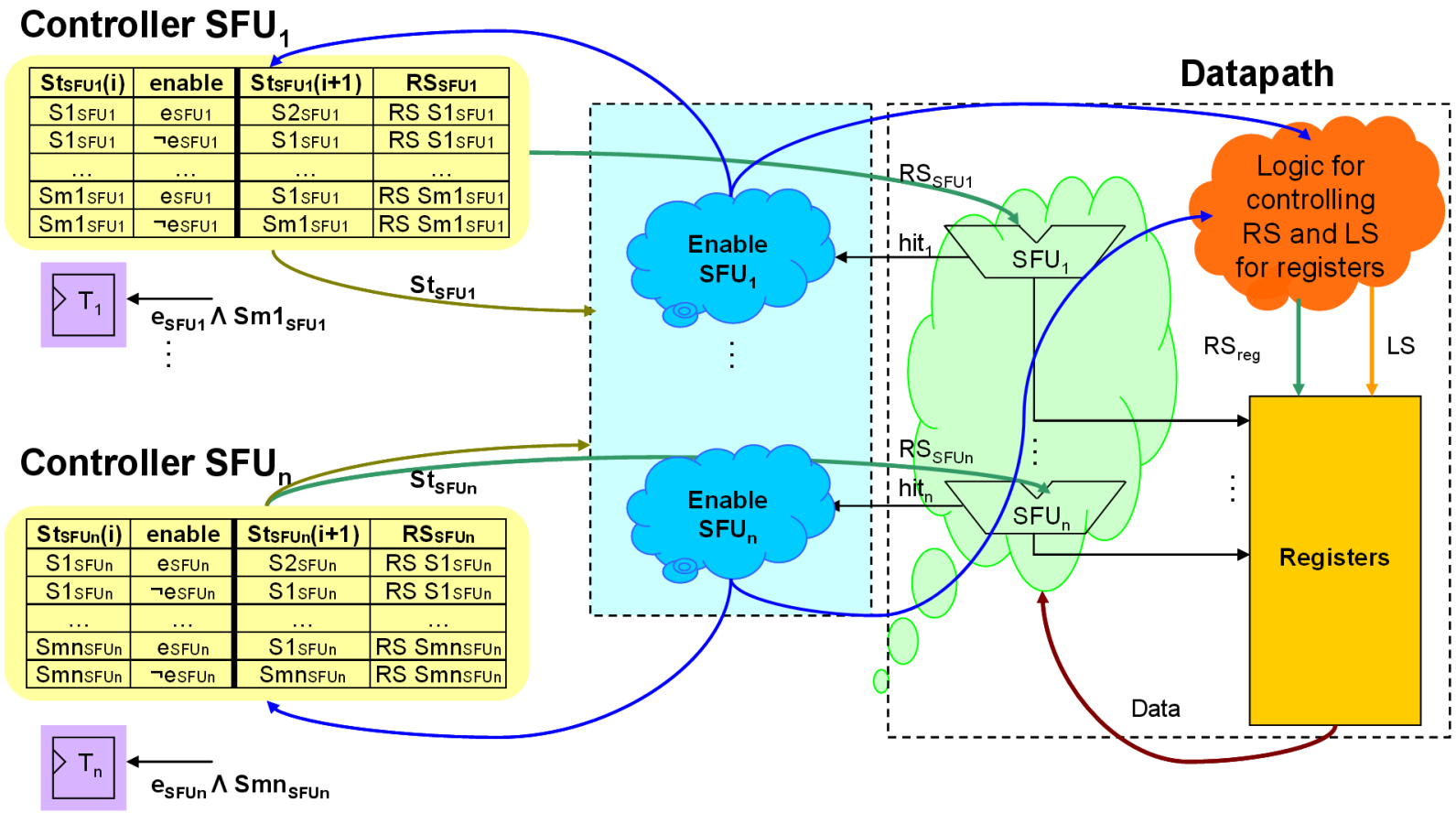


Figure 4.3: Distributed Management Architecture

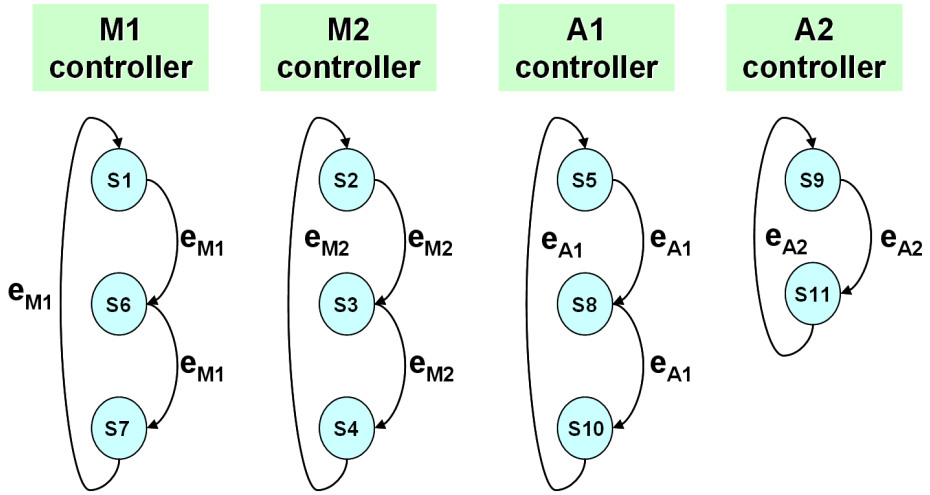


Figure 4.4: Local SFU controllers in the DiffEq benchmark

S4 belong to the same iteration than *Operation 6*, while S2 belongs to the following iteration. Therefore, the only *Compatible State* of M2 for *Operation 2* while *Operation 6* is being executed in M1, is S3. For a deeper explanation of how to calculate the *Compatible States*, the reader is invited to consult appendix B.

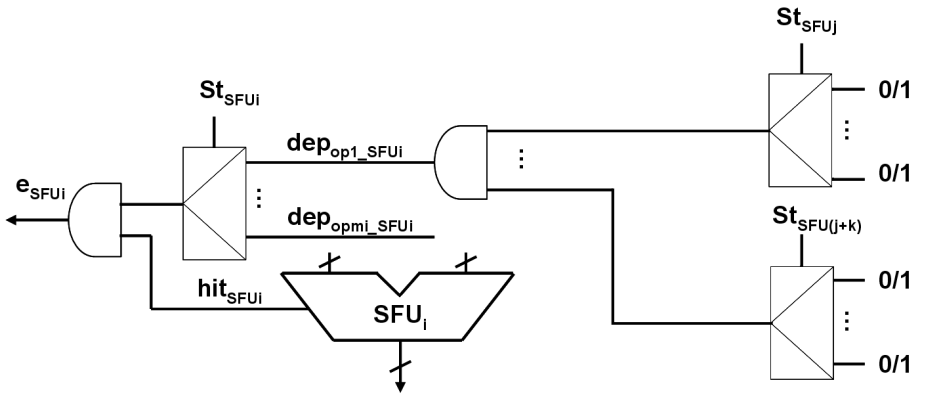
4.2.1. SFU Controllers

As it is depicted in figure 4.3, every SFU controller is a *Finite State Machine* (FSM) which generates the following state ($St_{SFU}(i+1)$) and the Routing Signals of the SFUs (RS_{SFU}), using the current state ($St_{SFU}(i)$) and the SFU enable signal (e_{SFU}).

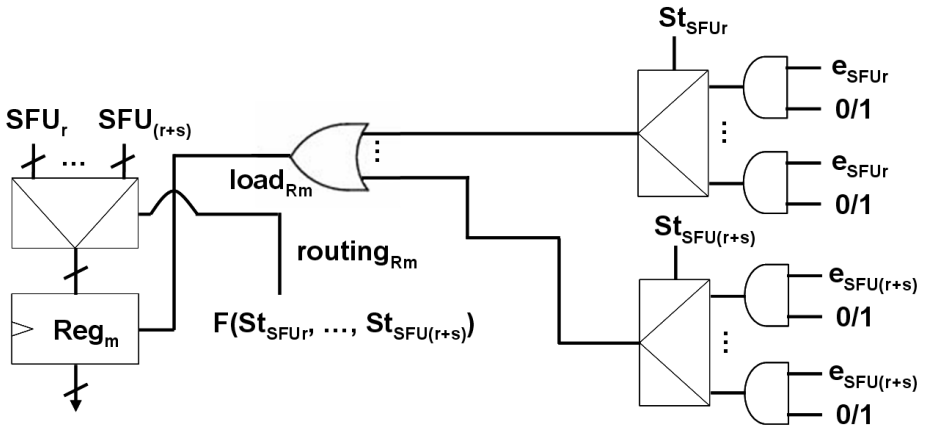
For example consider the DiffEq algorithm with the scheduling and binding shown in figure 4.1. The corresponding local controllers for every SFU are depicted by figure 4.4.

In every local SFU controller the states are labeled with the operation identifiers of the operations bound to the corresponding SFU, while the transitions, fired by the enable signals, are labeled with the SFU identifiers. Thus, M1 controller symbolic values are S1, S6 and S7 and transitions are fired by the enable e_{M1}.

These FSMs could be easily implemented with a counter modified everytime the corresponding e_{SFU} signal is fired.



(a) Enable Generation Unit Unit



(b) Register Routing and Load Signals generation

Figure 4.5: Enable Generation Unit and Register Signals generation

4.2.2. Enable Generation Units and Datapath control signals

After describing the SFU local FSMs that control the local states, the rest of signals which rule the datapath must be explained. In concrete, how to generate the enable SFU and Routing and Load Signals (LS) is a question that must be answered.

In figure 4.5 the canonical implementation of an EGU and the register Routing and Load Signals are depicted. First let's focus on the EGU, whose canonical implementation is shown in figure 4.5a. See how the three SEPT conditions are implemented with this logic. e_{SFUi} is the logic **AND** of the hit_{SFUi} signal and the output of a multiplexer. The hit signal corresponds with SEPT condition (1), while the output multiplexer is the logic **AND** of SEPT conditions (2) and (3). On the one hand, depending on the state (St_{SFUi}) only one input will be selected. This is SEPT condition (2). On

the other hand, multiplexer inputs are the values that will determine if the dependencies on every operation bound to SFU_i are solved or not. This is SEPT condition (3).

Every dependency line is the logic **AND** of several multiplexers output lines. These lines correspond with the individual dependencies that the operation executed in SFU_i has.

Now consider the SFU Routing Signals. These signals are generated by the corresponding local SFU controller. In fact, as there is a bijection between operations and states, the own local state will be enough for deciding what inputs must be led to the FU entries.

Secondly, let's see the Register Signals generation, which is depicted by figure 4.5b. The load signal is the logic **OR** of several multiplexer output lines. Every multiplexer identifies univocally the conditions to write every operation bound to the register. In order to fully identify an individual *enable operation signal*, it is enough to apply the logic **AND** function between the enable SFU signal of the SFU where that operation is bound, and the state of the corresponding local SFU controller. Hence, this behavior can be modeled with a multiplexer controlled by the state variable associated with the SFU that is executing the operation under question, and whose inputs are '0's or the own e_{SFU} signal.

In order to select what SFU result is going to be written in the register, a function of the local SFU states under question will be implemented. It is enough to build a truth table expressing the control signals of the multiplexer in terms of the SFU states corresponding to the operations bound to the register.

Note that the inputs to the dependencies lines and to the load generation line are outputs from multiplexers whose inputs are constants, i.e. '0' or '1'. The generation of these constants is the main task of this chapter and the most challenging task of this Ph.D. Thesis. Evidently this is only a canonical representation, and multiplexers will be optimized when synthesizing the circuits. However, the use of this representation helps to understand the architecture associated with the Distributed Management.

In the case of the dependencies, there will be a '1' value at the multiplexer entry iff an operation from a different SFU (SFU_j to $SFU_{(j+k)}$) is causing a dependency on the operation that is being executed in SFU_i. In the case of the load generation multiplexers, as the purpose is to determine what operation, bound to the SFU under question (SFU_r to $SFU_{(r+s)}$), is going to be written, there will be a '1' at an **AND** gate entry iff the operation from that SFU is bound to the register (Reg_m).

State	$routing_{R1}$
St_{M1}=S1	0
St_{M2}=S3	1
St_{M1}=S7	0

Table 4.2: Truth table for generating the control signal of the R1 multiplexer

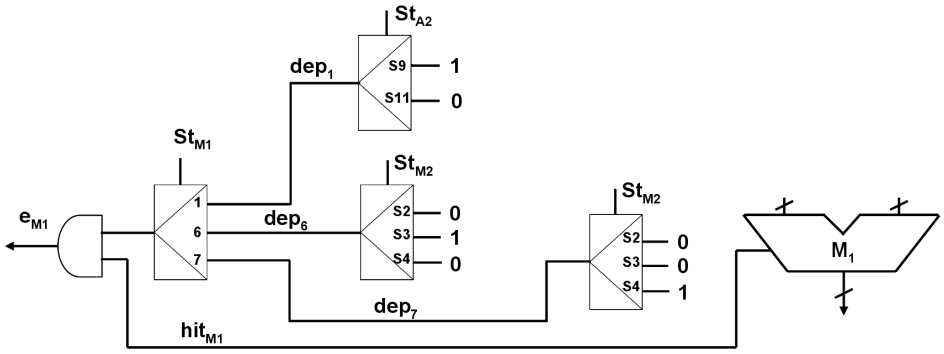
4.2.2.1. Application example

The best way to understand the canonic architecture of the Enable Generation Unit and the Registers Signals is to use an example, which is shown in figure 4.6. This figure depicts the implementation of the e_{M1} signal and the Routing and Load Signals for the register R1 in the DiffEq benchmark.

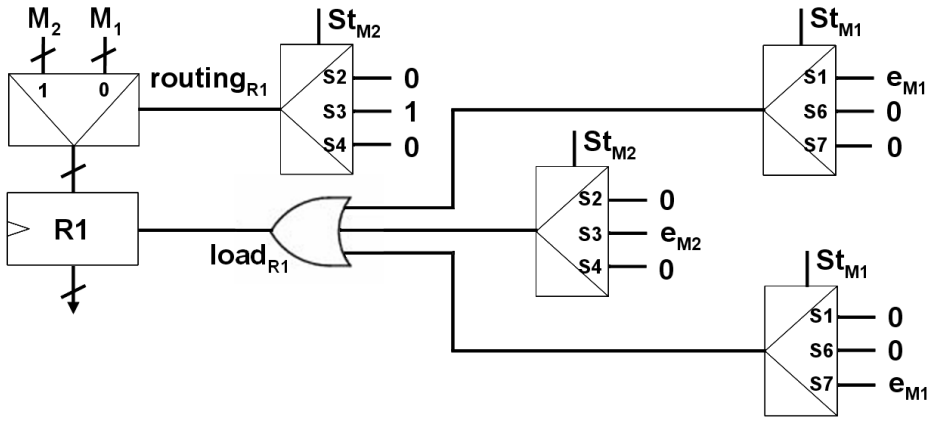
First let's consider e_{M1} generation, depicted by figure 4.6a, which is the logic **AND** of the hit_{M1} and the output of the multiplexer controlled by St_{M1}. This multiplexer has one entry per bound operation, where the dependency resolution lines arrive. Thus, in the entry labeled as 6, corresponding to *Operation 6*, the resolution line of the dependencies on *Operation 6* is connected. As it was explained at the end of subsection 4.1.2.1, it has a RAW and WAW hazard with *Operation 2*, which is executed in M2. Therefore the *Compatible States* of the local M2 controller, while executing *Operation 6* in M1, must be calculated. The only *Compatible State* is S3, so there is a logic '1' in the multiplexer entry labeled as S3. Note that this multiplexer is controlled by state St_{M2}, because the dependency is generated by an operation bound to M2. Note also that there is a RAW hazard with *Operation 1*, but as this operation is bound to M1 too, the own M1 state will indicate if it has been committed or not, and therefore it will not be necessary to take it into account in order to implement the *Operation 6* dependency resolution line. The same procedure is applied to *Operations 1* and *7*.

Now let's see register R1 signals generation, depicted by figure 4.6b. First note that *Operations 1*, *3* and *7* are bound to this register. In order to fully identify what operation must write the register, the e_{SFU} and the St_{SFU} signals are required. For example consider the upper input line of the **OR** gate, which is associated with *Operation 1*. e_{M1} is needed for knowing that an operation bound to M1 is going to write, while the St_{M1} value will indicate that *Operation 1*, and not *Operations 6* or *7*, is the one bound to M1 whose result will be written.

Finally consider how to select the SFU which will write in R1. A truth table must be built in order to determine the control signals of the multiplexer that is at the input of the register. See table 4.2. Without loss of generality, it can be supposed that M1 result will be at the multiplexer entry labeled as 0, and M2 result at the entry labeled as 1. Hence, the entry labeled as 1 will only be utilized if St_{M2}=S3, otherwise the 0^{th} entry will be selected.



(a) DiffEq M1 Enable Generation Unit Unit



(b) DiffEq R1 Routing and Load Signals generation

Figure 4.6: DiffEq Enable Generation Unit and Register Signals generation

Therefore St_{M2}=S3 is the condition that will implement the control signal of the multiplexer.

The pseudocode of the procedures followed to generate the datapaths, in a similar fashion to the example, is shown in algorithms 4.2 and 4.3. If n and $\overline{H}$ are the number of operations in the DFG and the average number of hazards per operation, respectively, then the complexity of algorithm 4.2 is $\mathbf{O}(\overline{H}n^3)$, provided that the complexity of *getCompatibleStates* is $\mathbf{O}(\overline{H}n)$ (see appendix B). On the other hand, if $\overline{r}$ is the mean number of operations bound to a register, the complexity of algorithm 4.3 is $\mathbf{O}(\overline{r})$.

4.2.3. Iterations control

In addition to the *Compatible States*, for verifying the SEPT properly, the CSLU also needs to know in which iteration an operation is. In order to check the iteration, one T-flipflop per SFU has been included into every local controller. T-flipflops will toggle when the last operation bound to the

Algorithm 4.2 generateEGU(FU_e)

```
1: depMultiplexers ← {}
2: depAnds ← {}
3: for all O_e bound to FU_e do
4:    CS ← {}
5:    for all O_d → O_e do
6:       CS ← CS ⋃ getCompatibleStates(St_e, St_d)
7:    end for
8:    MCS ← getMinimumCompatibleStatesSet(CS)
9:    for all St_d ∈ MCS do
10:      newMultiplexer ← generateMultiplexer(St_d)
11:      depMultiplexers ← depMultiplexers ⋃ newMultiplexer
12:   end for
13:   newAnd ← generateAnd(depMultiplexers)
14:   depAnds ← depAnds ⋃ newAnd
15: end for
16: Mux_e ← generateMultiplexer(St_e, depAnds)
17: generateAnd(Mux_e, FU_e)
```

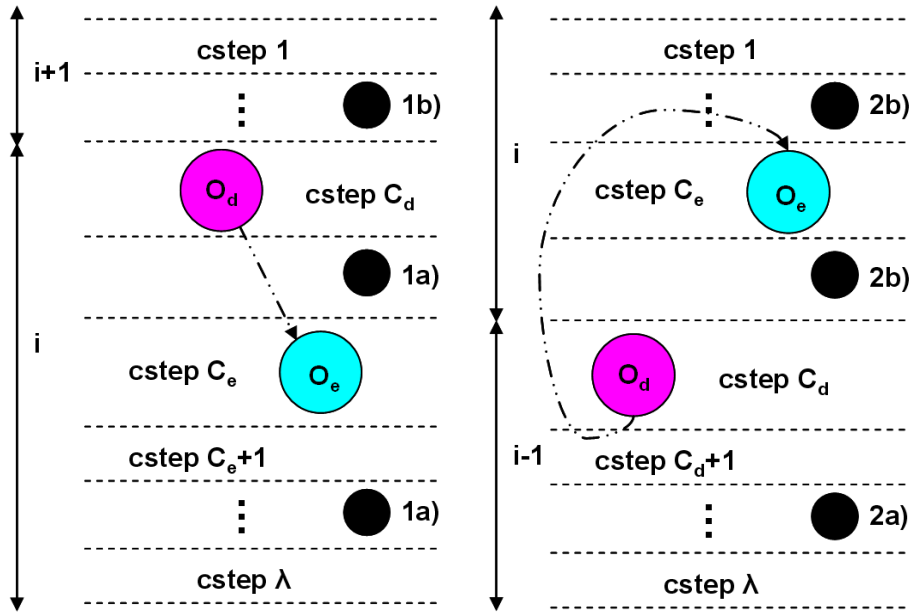
Algorithm 4.3 generateRegisterControl(R_e)

```
1: loadMultiplexers ← {}
2: for all O_e bound to R_e do
3:    newMultiplexer ← generateMultiplexer(St_e, FU_e)
4:    loadMultiplexers ← loadMultiplexers ⋃ newMultiplexer
5: end for
6: load ← generateOr(loadMultiplexers)
7: ctrlMux ← generateControl(loadMultiplexers, R_e)
8: regMultiplexer ← generateMultiplexer(R_e, ctrlMux)
```

SFU within an iteration is committed.

For example consider *Operation 5* in figure 4.1. Suppose that it has been bound to a new adder A3. *Operation 8* has a RAW hazard, so in order to solve it we must check St_{A3} ">" S5, but although *Operation 5* were committed, the next state would be S5, too. Therefore this *new* S5 state should be distinguished from the previous S5 state. Hence, information about the current iteration is required.

Theorem 3 (Iterations Theorem)**.** *Let O_e be an operation. Let O_d be an operation with a dependency with O_e. Let O_{comp} be an operation whose bound SFU is in a compatible state with the SFU state of O_e. Finally, let C_e, C_d and C_{comp} be the csteps where the respective operations have been statically scheduled. If O_e is in iteration i, there will be the following cases:*



(a) Iterations Theorem, case 1 (b) Iterations Theorem, case 2

Figure 4.7: Iterations Theorem cases

(1) $C_d \leq C_e$

- *(a)* $C_{comp} > C_d$. *Then* O_e, O_d *and* O_{comp} *are in the same iteration* i
- *(b)* $C_{comp} \leq C_d$. *Then* O_{comp} *is in iteration (i+1) while* O_e *and* O_d *are in iteration* i

(2) $C_d \geq C_e$

- *(a)* $C_{comp} > C_d$. *Then* O_d *and* O_{comp} *are in iteration (i-1) while* O_e *is in iteration* i
- *(b)* $C_{comp} \leq C_d$. *Then* O_e *and* O_{comp} *are in iteration* i*, while* O_d *is in iteration (i-1)*

Proof. First, we observe that according to the calculation of *Compatible States*, O_e and O_{comp} will be separated by at most one iteration. Therefore, using one T-flipflop is sufficient.

Second, O_d and O_{comp} are executed in the same SFU. It should be noticed that this happens because of the *Compatible States* definition. Therefore, it is easy to verify that if $C_d < C_{comp}$ then, they are in the same iteration, and if $C_d \geq C_{comp}$ then O_{comp} is in the immediately following iteration, because O_{comp} is always executed after O_d.

Next, see figure 4.7 for an illustration of these relationships. Consider case (1). If $C_d \leq C_e$, then, O_d and O_e must be in the same iteration i. If

(1a) holds, then O_{comp} must be in the same iteration as O_d (i), while if (1b) holds it must be in the following iteration *(i+1)*.

In case (2), if $C_d \geq C_e$, then O_d must be one iteration behind of O, that is *(i-1)*. If (2a) holds, $C_{comp} > C_d$, then, O_{comp} and O_d must be in the same iteration (*i-1*). If (2b) holds, $C_{comp} \leq C_d$, then O_{comp} must be in the next iteration, that is *(i-1+1)* = *i*.

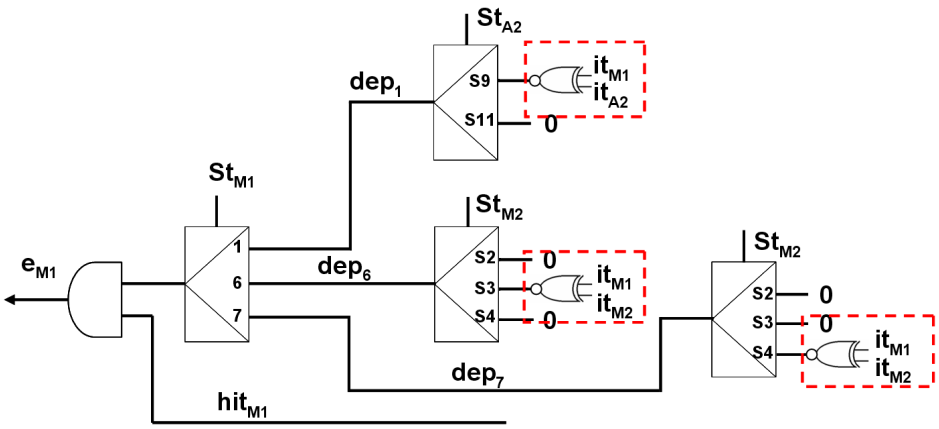
Note that the equality sign is used in both cases (1) and (2). In case (1), equality is only valid for WAR dependencies, while in case (2) equality is only valid for WAW dependencies. The reason is that two operations with WAR dependencies can be scheduled in the same cycle, so O_e and O_d must be in the same iteration *i*. On the other hand, since operations O_d causing WAW dependencies are statically scheduled in prior cycles to O_e, if $C_d = C_e$ then O_d must be in the previous iteration (*i-1*). In fact, if $C_d = C_e$, (since the binding is static) it is due to $O_d = O_e$. Finally, note that the equality case is not possible for RAW dependencies, as operation O_d is always scheduled prior to O_e. □

Therefore, according to the Iteration Theorem above, it is sure that at any given time two T-flipflops can only have the same value if their respective SFUs are in the same iteration. Since the controller only needs to keep account of iterations that can be maximally one iteration apart, there is no need for a large counter and a T-flipflop is sufficient. Note that if two operations have no dependencies they will always be able to be executed simultaneously. For example consider the extreme case where there is a DFG with two independent operations. If there are several failures in one and no failures in the other, since there are no dependencies, they can run independently separated by several iterations.

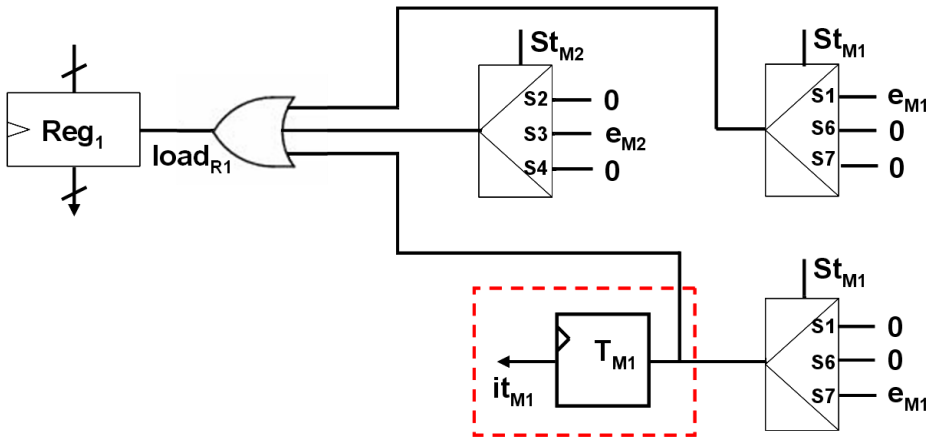
4.2.3.1. Application example

The inclusion of the information given by the T-flipflops, i.e. iterations, must be taken into account in order to generate the datapaths. In this subsection the impact of this information will be taken into account in the example previously shown by figure 4.6.

Then, see figure 4.8a. As it is observed, an **XNOR** gate has substituted the previous logic '1' values at the entry of the dependency multiplexers. They are surrounded by a dashed line. This gate is applied over a couple of signals labeled as it_{SFU}, which are the outputs of the corresponding SFU T-flipflop. For example consider *Operation 6*, its dependency produced by *Operation 2* and its *Compatible State* S3, which corresponds with *Operation 3*. *Operations 6* and *3* have been statically scheduled in cstep 2, while *Operation 2* in cstep 1. Then as RAW and WAW hazards are being considered, and $C_d \leq Ce$, we are in case (1) of the Iterations Theorem. As $C_{comp} > C_d$, we are in case (1a), i.e. *Operations 2*, *3* and *6* are in the same iteration, which



(a) DiffEq M1 Enable Generation Unit



(b) DiffEq R1 Load Signals generation

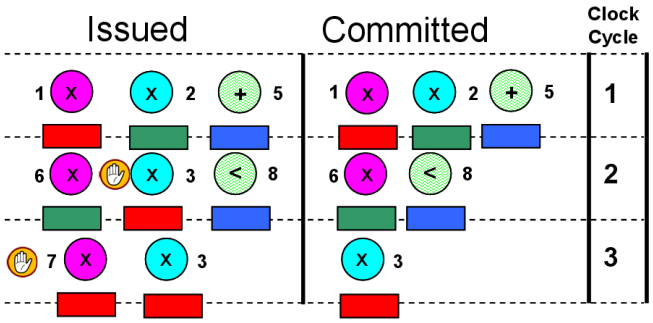
Figure 4.8: DiffEq Enable Generation Unit and Register Load Signals generation with iterations information

is implemented with an **XNOR** gate. If they were in different iterations, an **XOR** gate would be used instead.

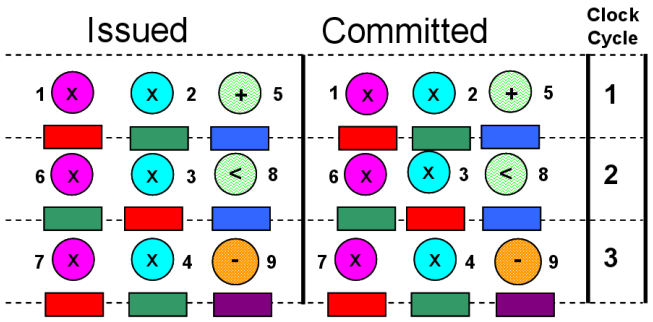
Finally consider the T-flipflop included in figure 4.8b for completing the M1 EGU. It must toggle every time the last operation bound to M1 is committed, i.e. *Operation 7*. This is identified with the logic **AND** of e_{M1} and St_{M1}=S7, which was utilized in the R1 load signal generation.

4.2.4. Circular Dependencies

Let's consider now *Operation 3* in the DiffEq example. It can be committed if hit_{M2}='1', St_{M2}=S3, and if M1 is in a *Compatible State* with *Operation 3*, because of the WAR hazard with *Operation 6*, and because of the WAW dependency between *Operations 1* and *3*. The list of *Compatible States* for



(a) Wrong execution flow



(b) Right execution flow

Figure 4.9: WAR circular dependencies in the DiffEq execution flow

M1 because of the WAR hazard, CS_{WAR}, is {S7} and because of the WAW hazard, CS_{WAW}, is {S6, S7}. In other words, the WAR hazard with *Operation 6* is avoided if St_{M1}=S7, and the WAW hazard with *Operation 1* if St_{M1}=S6 or St_{M1}=S7. Therefore the final dependency condition will be St_{M1}=S7, which is in the same iteration according to the Iterations Theorem. In these conditions, due to the WAR hazard, *Operation 3* will only be committed after *Operation 6* is committed, i.e. when St_{M1}=S7. This forces *Operation 3* to remain stalled for one cycle, which is shown in figure 4.9a. In order to solve this problem, the WAR condition St_{M1}=S7 must take into account that *Operation 3* can be committed if *Operation 6* is committed, and then, the datapath will work properly, as depicted in figure 4.9b.

Thus, in the case of WAR dependencies the enable signal of the operation that is causing the hazard (e_{M1}, due to *Operation 6*) must be included in the enable condition of the operation under question (e_{M2}, due to *Operation 3*). This does not happen with RAW or WAW dependencies because they cannot be committed simultaneously, as the operations that generate the dependency are statically scheduled in previous cycles.

However, the use of enable signals to generate yet another enable signal can cause a problem. It is possible to encounter a situation where in order

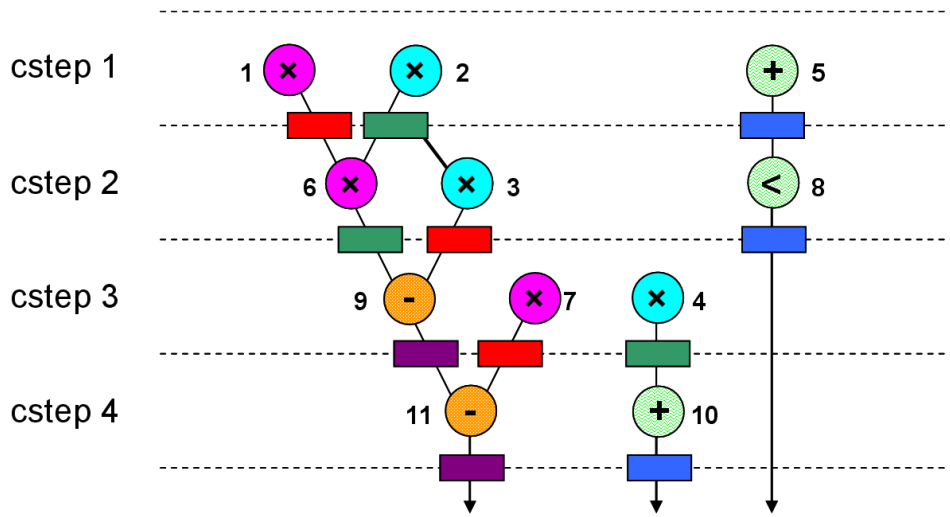


Figure 4.10: DiffEq example with additional edge between Operations 2 and 3

to commit operation X_1, X_2 must be committed, ..., to commit operation X_N, X_1 must be committed. This circular dependency or *WAR cycle* will produce a deadlock if all involved operations are executed at the same time.

Theorem 4 (*WAR Cycles Theorem* (WARCT))**.** *WAR cycles only occur between operations that are statically scheduled in the same cycle.*

Proof. Let O and O_{war} be two operations that suffer a WAR cycle. Then, let R be the register that O must write and O_{war} must read, and R_{war} the register that O_{war} must write and O must read. And let C_o and C_{war} be the csteps where they have been statically scheduled. As O_{war} must read R before O writes it, O_{war} must be committed earlier than or in the same cycle as O. Since O must read R_{war} before O_{war} writes it, O must be committed earlier than or in the same cycle as O_{war}. So, both O and O_{war} must be committed in the same cycle. Suppose they are statically scheduled in different csteps and $C_{war} < C_o$, without loss of generality. Therefore, there would be a static inconsistency in the circuit, as O_{war} would be writing R_{war} before O could read it, which is not possible because the static scheduling would not allow it. □

Thanks to the WARCT we know where deadlocks occur, but a mechanism must be developed to avoid them. In order to illustrate this, let's consider a new example, shown in figure 4.10. This DFG is the same as the DiffEq example with only one additional edge between *Operations 2* and *3*. The introduction of this edge causes a circular WAR dependency between *Operations 3* and *6*. Therefore, enable signals must be modified.

The problem is that conditions must work both when there is and there is not a WAR cycle (for example a reference from another operation not involved in the own WAR cycle). In order to achieve this, every enable reference, which belongs to a WAR cycle, is substituted by a *pseudo-enable* reference. A pseudo-enable is almost the same condition as the original enable, but only without the WAR condition that contains the enable reference causing the WAR cycle. Equation 4.1 depicts a situation where a WAR cycle happens between certain *Operations 1* and *2*. Suppose then that *e1* and *e2* are the values of e_{SFU1} and e_{SFU2} when *Operations 1* and *2* are going to be committed by the controllers of *SFU1* and *SFU2*, respectively. *e1* references *e2* in the WAR condition ($war1_{e2}$) and *e2* references *e1* in its WAR condition ($war2_{e1}$) too. Then, *e1* and *e2* will be substituted by *pseudo_ e1* and *pseudo_ e2* in the $war2_{e1}$ and $war1_{e2}$ signals, respectively.

$$\begin{aligned} e1 &<= hit_{SFU1} \wedge St_{SFU1} \wedge raw1 \wedge waw1 \wedge \\ &\quad (war1_1 \wedge \ldots war1_{e2} \ldots \wedge war1_M) \\ war1_{e2} &<= e2 \vee \ldots \\ e2 &<= hit_{SFU2} \wedge St_{SFU2} \wedge raw2 \wedge waw2 \wedge \\ &\quad (war2_1 \wedge \ldots war2_{e1} \ldots \wedge war2_N) \\ war2_{e1} &<= e1 \vee \ldots \end{aligned} \tag{4.1}$$

Equation 4.2 shows the pseudo-enable definition

$$\begin{aligned} pseudo\_e1 &<= hit_{SFU1} \wedge St_{SFU1} \wedge raw1 \wedge waw1 \wedge \\ &\quad (war1_1 \wedge \ldots \wedge war1_M) \\ pseudo\_e2 &<= hit_{SFU2} \wedge St_{SFU2} \wedge raw2 \wedge waw2 \wedge \\ &\quad (war2_1 \wedge \ldots \wedge war2_N) \end{aligned} \tag{4.2}$$

and finally equation 4.3 is the result of substituting the enable signals, that were causing the WAR cycle in equation 4.1, by the pseudo-enable signals defined in equation 4.2.

$$\begin{aligned} e1 &<= hit_{SFU1} \wedge St_{SFU1} \wedge raw1 \wedge waw1 \wedge \\ &\quad (war1_1 \wedge \ldots war1_{e2} \ldots \wedge war1_M) \\ war1_{e2} &<= pseudo\_e2 \vee \ldots \\ e2 &<= hit_{SFU2} \wedge St_{SFU2} \wedge raw2 \wedge waw2 \wedge \\ &\quad (war2_1 \wedge \ldots war2_{e1} \ldots \wedge war2_N) \\ war2_{e1} &<= pseudo\_e1 \vee \ldots \end{aligned} \tag{4.3}$$

4.2.4.1. Application example

In the example of figure 4.10, the values of the enable signals e_{M1} and e_{M2} when *Operations 6* and *3* are going to be committed, respectively, are expressed as follows:

$$\begin{aligned} e6 <= hit_{M1} \wedge St_{M1} = S6 \wedge (e3 \vee (St_{M2} = S4 \wedge it_{M1} = it_{M2})) \\ e3 <= hit_{M2} \wedge St_{M2} = S3 \wedge (e6 \vee (St_{M1} = S7 \wedge it_{M1} = it_{M2})) \end{aligned} \tag{4.4}$$

This will be transformed into

$$\begin{aligned} e6 <= hit_{M1} \wedge St_{M1} = S6 \wedge (pseudo\_e3 \vee (St_{M2} = S4 \wedge it_{M1} = it_{M2})) \\ e3 <= hit_{M2} \wedge St_{M2} = S3 \wedge (pseudo\_e6 \vee (St_{M1} = S7 \wedge it_{M1} = it_{M2})) \\ pseudo\_e6 <= hit_{M1} \wedge St_{M1} = S6 \wedge (St_{M2} = S4 \wedge it_{M1} = it_{M2}) \\ pseudo\_e3 <= hit_{M2} \wedge St_{M2} = S3 \wedge (St_{M1} = S7 \wedge it_{M1} = it_{M2}) \end{aligned} \tag{4.5}$$

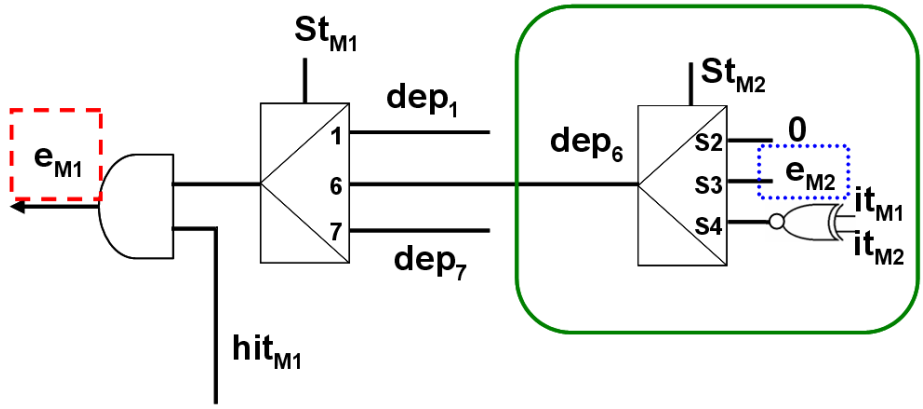
And finally as *Operations 3* and *6* must be committed simultaneously, according to the WARCT, the iterations condition can be removed and then the enables values can be simplified to

$$\begin{aligned} e6 <= hit_{M1} \wedge St_{M1} = S6 \wedge pseudo\_e3 \\ e3 <= hit_{M2} \wedge St_{M2} = S3 \wedge pseudo\_e6 \\ pseudo\_e6 <= hit_{M1} \wedge St_{M1} = S6 \\ pseudo\_e3 <= hit_{M2} \wedge St_{M2} = S3 \end{aligned} \tag{4.6}$$

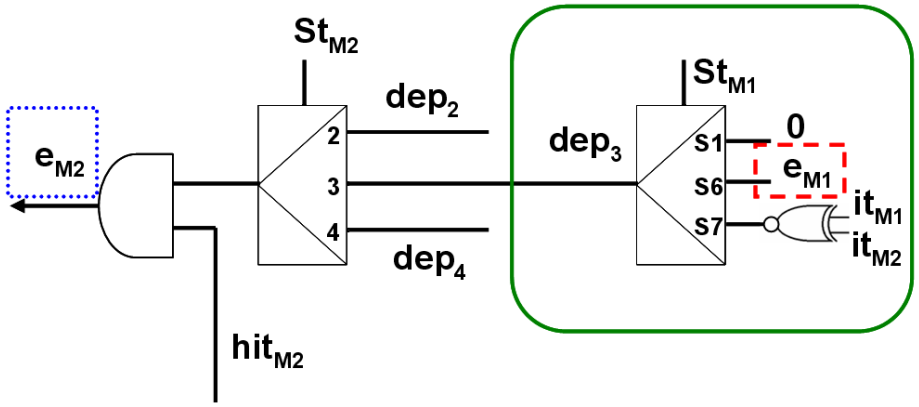
Note that this is only the logic transformation that must be made at the operation level. In order to avoid the WAR cycles, every involved operation must implement the conditions shown in the previous equations. However, the reader must remember that the Distributed Management architecture proposed in section 4.2 was defined with one enable signal per SFU, so some changes must be done in order to control these situations. In concrete, the dependency lines of the affected operations will be modified.

Considering the example of the DiffEq benchmark with the additional edge between *Operations 2* and *3*, as shown in figure 4.10, and taking into account the principles and architecture described in this chapter, previously to equation 4.6, the implementation of both e_{M1} and e_{M2} enable SFU signals would be as depicted by figure 4.11.

See figure 4.11a. First, the reader must notice that with this new example *Operation 6* will be committed if *Operation 3* is committed (e_{M2} **AND** St_{M2}=*S3*), or if it has been already committed (St_{M2}=*S4* **AND** (it_{M1}



(a) DiffEq M1 Enable Generation Unit

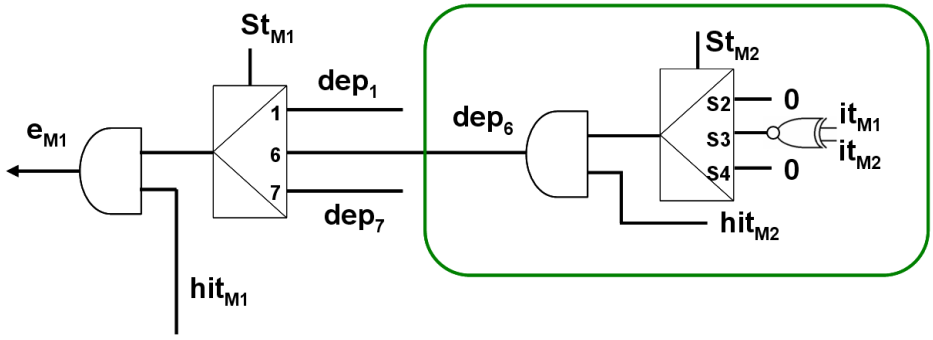


(b) DiffEq M2 Enable Generation Unit

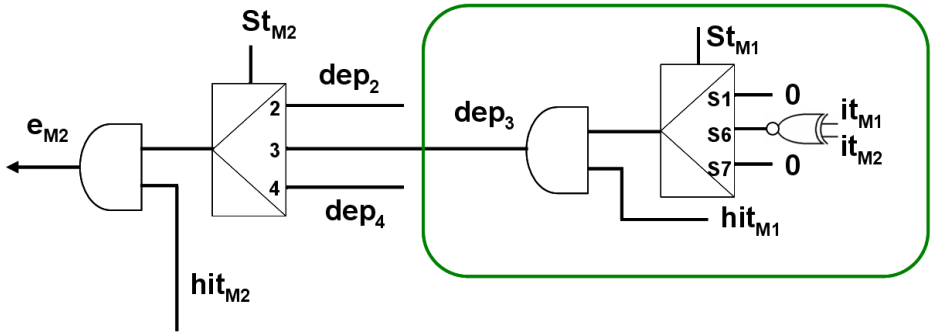
Figure 4.11: WAR cycle implementation in the DiffEq example with additional edge between *Operations 2* and *3*

XNOR it_{M2})). Now, see figure 4.11b. Similarly, *Operation 3* will be committed if *Operation 6* is committed (e_{M1} **AND** St_{M1}=*S6*) or if it has been already committed (St_{M1}=*S7* **AND** (it_{M1} **XNOR** it_{M2})). This fact has been shown in equation 4.4.

As it can be observed in the dep_6 signal generation, e_{M1} depends on e_{M2}. Analogously, in the dep_3 signal it can be seen that e_{M2} depends on e_{M1}. This combinational bow must be broken, so both dep_6 and dep_3 signals implementations, surrounded by a solid line in figure 4.11, are substituted by the implementations shown by figures 4.12a and 4.12b, which correspond with equation 4.6.



(a) DiffEq M1 Enable Generation Unit



(b) DiffEq M2 Enable Generation Unit

Figure 4.12: Pseudo-enables implementation in the DiffEq example with additional edge between *Operations 2* and *3*

4.2.5. Updating the SFU Predictors

Finally, the last question remaining is when the SFU predictor tables should be updated. At first glance and based on data locality, it could seem they should be updated when there is a failure but the operation has resolved its dependencies and it is ready for being committed (except for the failure). However, this may not be possible because of the aforementioned deadlocks. If none of the executed operations is committed, and there are WAR or WAW dependencies between them, it is possible that predictors cannot be updated because none of the operations is ready for being committed. Hence, there would appear a deadlock.

Another fact that must be taken into account is that, due to dependencies, operations frequently have to wait to be committed, but they can use these stall cycles for being corrected, thereby improving performance. On the other hand, there are situations when some of the operands will keep previous values until the new values are written in the corresponding registers. This will lead to *unexpected* operations and thus predictor tables could become dirty with these *unexpected values*. Nevertheless, the second case is much less

frequent that the first one because of data correlation. In spite of operating with *unexpected* values, if they are similar to the *expected* operands, they both will produce similar carries. Hence, when the *expected operands* arrive, the prediction will produce a hit. Thereby, it is better to always update predictor tables than to update them only if RAW dependencies have been resolved.

4.3. Multicycling and chaining

The inclusion of multicycle SFUs and the chaining technique will be studied in this section. On the one hand multicycle SFUs will help to boost performance as in the case of Centralized Management, because the penalty due to failures will be less significant than in the case of monocycle SFUs. On the other hand, the chaining technique will try to reduce the number of penalty cycles by eliminating some hazards.

4.3.1. Multicycle SFUs

As in the case of Centralized Management, using multicycle SFUs will increase performance, mainly because penalty cycles will be less significant with respect to the total number of cycles required to calculate an operation, according to the timing analysis performed in section 2.3. Moreover, the indirect consequence of multicycle long latency FUs will be that most of short latency FUs failures will be hidden because otherwise these modules would be probably waiting anyway for the long latency FUs to finish. For example consider the case of an addition and a product being executed simultaneously in their respective SFUs, such that the next addition bound to the adder must wait the product to finish. Besides consider that the speculative adder and multiplier have a latency of 1 and 3 short cycles respectively. If the addition fails it will be executed in 2 short cycles, while the product will take 3 short cycles anyway. The next addition will be executed after the 3 short cycles of the product, as if no failure had happened. On the other hand, if a monocycle implementation is considered, the product would take 1 long cycle and the addition 2, stalling thus the next addition bound to the adder and penalizing performance. Furthermore, multicycle SFUs are more beneficial with Distributed Management than with Centralized Management. Using Centralized Management the whole datapath would stall 1 cycle because of the failure, so both operations would take 4 cycles, instead of the 3 cycles that these operations last with the Distributed Management implementation.

With Centralized Management everything is simpler, because there is a global controller that guarantees that in a determined state an operation is finished, i.e. committed. This is not the case of Distributed Management. There could be operations that fail or operations whose operands are not

Q(i)	e_{SFU}	hit_{SFU}	RAW_{SFU}	Q(i+1)	Load	Count
>0	0	-	0	Q(i)	0	0
>0	0	-	1	Q(i)-1	0	1
=0	0	1	1	0	0	0
=0	0	0	1	$C_{Pen}-1$	1	-
=0	1	1	1	$\lambda_{SFU}-1$	1	-

Table 4.3: Multicycle SFU counter truth table

correct because they are waiting for a previous result. And what is more, with multicycle SFUs there could be operations calculated with correct operands and producing a '1' in the hit_{SFU} signal, but that have not finished their calculation time, so result is not valid yet. As hit_{SFU} is a combinational signal, there could be a spurious signal such that it becomes a logic '1' while the number of cycles used for a calculation is lower than the SFU latency. Hence, additional hardware must be designed carefully for controlling all these situations.

Table 4.3 is a summary of the aforementioned cases. Let's suppose that a generic SFU takes λ_{SFU} cycles in computing a result with a correct prediction, and needs C_{Pen} additional cycles for recalculating the result if there is a misprediction. A counter Q will be necessary in order to keep the information of the calculation cycle, and thereby the *Load* and *Count* signals associated to the counter will need to be implemented too. In other words, table 4.3 is the truth table of these three signals: Q, *Load* and *Count*.

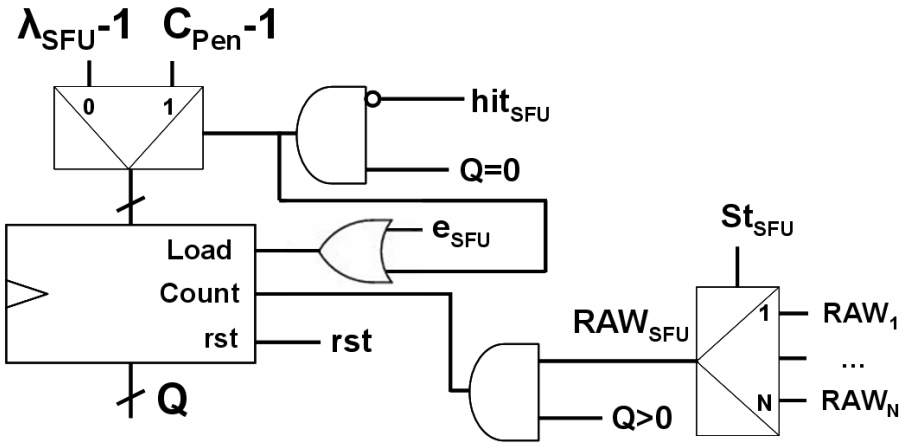
The idea is to use the counter to load either λ_{SFU}-1 or C_{Pen}-*1*, and count iff the operands are correct. Then, when Q=0 it is possible to decide whether or not the calculation or the correction period has finished, taking into account other signals too. Hence Q=0 is a signal that must be included in the e_{SFU} generation. Note that as the count will be produced if RAW hazards have been solved, the e_{SFU} and hit_{SFU} signals will be enough for deciding if to count or load a new value. Hence, let's examine every case of table 4.3.

(1) Q>0. It is not possible to commit the operation because the calculus or correction is still being performed.

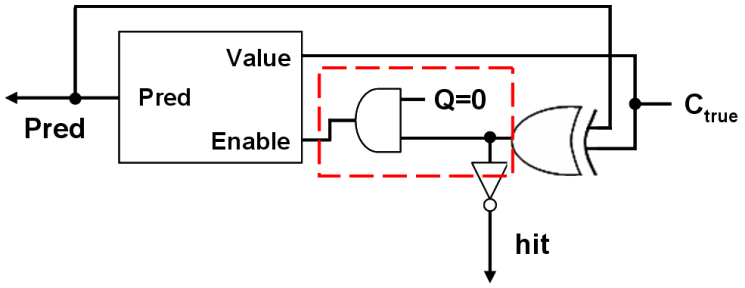
  (1.1) RAW_{SFU}=0. The operands are not correct. Hence we do not count. Note that this does not mean that the SFU stops calculating with *unexpected* values.

  (1.2) RAW_{SFU}=1. The operands are correct but the operation has not finished yet. Therefore, the calculus or correction must continue its execution.

(2) Q=0. Then it is possible that the operation is ready to be committed.



(a) Cycle counter and glue logic for incorporating Multicycle SFUs to the Distributed Management Architecture



(b) Modification in the predictor of a multicycle SFU

Figure 4.13: Modifications for using multicycle SFUs with Distributed Management

(2.1) e_{SFU}=0. Then the operation cannot be committed.

(2.1.1) hit_{SFU}=0. There is a misprediction. C_{Pen}-1 must be loaded.

(2.1.2) hit_{SFU}=1. There is a hit, but due to a WAR or WAW hazard the operation cannot be committed. Q=0 must be maintained.

(2.2) e_{SFU}=1. Then the operation can be committed. Note that e_{SFU}=1 implies that Q=0. λ_{SFU}-1 must be loaded for the following operation.

Figure 4.13a shows the general structure that needs to be included in order to incorporate SFUs to the Distributed Management architecture. There is a counter whose *Load* signal is activated everytime an operation is committed (e_{SFU}) or everytime there is a misprediction (the **AND** output). When the system is reset (*rst*) the counter will be reset too. The *Count* signal is activated only if the RAW hazards of the corresponding operation are solved

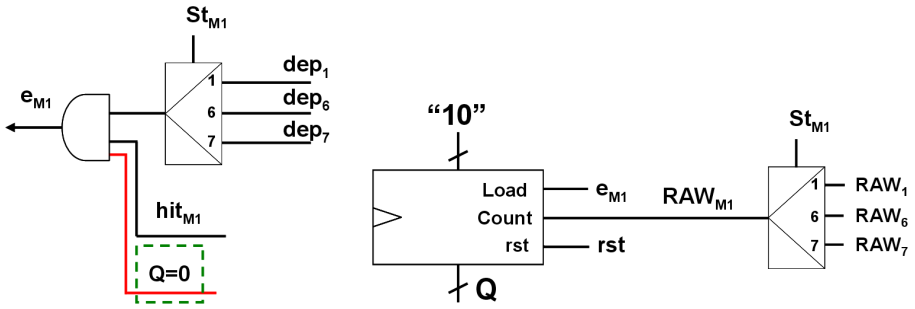


Figure 4.14: Cycle counter and glue logic for the M1 controller in the DiffEq example

and if we must count (Q>0). The RAW_1, ..., RAW_N signals are the same than the ones that compose the dependencies resolution lines in the SFU controller.

The C_{Pen} value will be loaded only if Q reaches the '0' value, i.e. the RAW hazards have been solved, and if there is a misprediction. In any other case, the λ_{SFU} value will be driven to the counter input. Finally, note that it is necessary to subtract 1 unit to these values, in order to count exactly λ_{SFU} or C_{Pen} cycles.

In addition to this, an **AND** gate must be included for enabling the writing of the predictor utilized in the SFUs only if the calculation or correction time has finished, i.e. Q=0. This is depicted in figure 4.13b.

In conclusion, the introduction of multicycle SFUs produces a negligible area overhead with respect to the monocycle implementation, because only some small counters and glue logic need to be added. On the other hand, in the case of a traditional controller, as in the conventional or the Centralized Management implementations, the use of multicycle FUs implies an increase in the number of states. This states increase can be illustrated by the difference between figures 4.1 and 3.5, where 4 and 10 states are required to implement the controller, respectively. Moreover, it must be noted that figure 3.5 corresponds with the best case scheduling with SFUs. As non-speculative FUs need more cycles, the conventional implementation would require 14 csteps or states to complete an iteration of the DiffEq example.

4.3.1.1. Application example

In this subsection the multicycle counter for the M1 controller in the DiffEq example will be shown. The reader must take into account the time model of this SFU, which was explained in section 2.3. Thus, a Predictive Multiplier will take 3 cycles if it hits and 4 if it does not. Therefore λ_{M1}=3 and C_{Pen}=1.

As C_{Pen}=1, the glue logic will be simplified. See figure 4.14. There is no need to use a multiplexer to decide the input value to the counter. Everytime there is a misprediction, the correct result will always be produced in the following cycle, when the hit_{M1} signal will change to '1'. Finally note how Q=0 is incorporated to the **AND** that generates the e_{M1} signal.

4.3.2. Chaining

As it has been explained in section 3.3.2, chaining allows the execution of several operations with data dependencies in the same cstep. The main reason for using chaining with SFUs is that it can reduce the number of stall cycles.

The number of hazards will be reduced, as depicted by figure 4.15. Let's suppose that a certain operation O_d writes a register R_d that must be read by operation O_e, i.e. O_d causes a RAW hazard to O_e (RAW_{Ode}). Besides there could be hazards due to R_d with other operations (WAR_{Od} and WAW_{Od}). On the other hand, O_e may also have dependencies. There is a RAW hazard with O_d (RAW_{Ode}), and possibly there could be other RAW hazards with other operations ($\text{RAW}_{\text{Oe}} \backslash \text{RAW}_{\text{Ode}}$). In addition to this, there also exist the dependencies due to R_e (WAR_{Oe} and WAW_{Oe}). After applying chaining evidently RAW_{Ode} will disappear, so O_d and O_e can be executed simultaneously, but also WAR_{Od} and WAW_{Od}. This fact will eliminate dependencies with later operations, that with Distributed Management could be executed and committed if their other hazards were solved.

Note that this does not happen with Centralized Management, because using chaining will not take advantage of the removal of the R_d writing performed by O_d. The following operations that had a dependency, because without chaining R_d was being written by O_d, will have to wait anyway until the cstep where they had been statically scheduled.

From the architectural point of view, as the subset of operations that are chained must be committed simultaneously, the signals that will fire the state transitions will be the logic **AND** of the enables that fired them without chaining. Hence, the Enable Generation Unit and the Registers Signal generation will remain the same. However, the reader may note that with chaining there will be less written operations, so the registers Load and Routing Signals must be modified, of course, but this is due to a change in the Dataflow Graph, not in the datapath generation procedure.

Finally, the update of the T-flipflop will be produced as usual, i.e. when the last operation bound to the SFU under consideration is committed.

One last question that must be evaluated is the compatibilty of chaining with multicycle SFUs. If chaining is applied, cycle time will increase. However, as linear structures, such as Ripple Carry Adders, are being utilized, this increase will be negligible if only few operations are chained, as explained

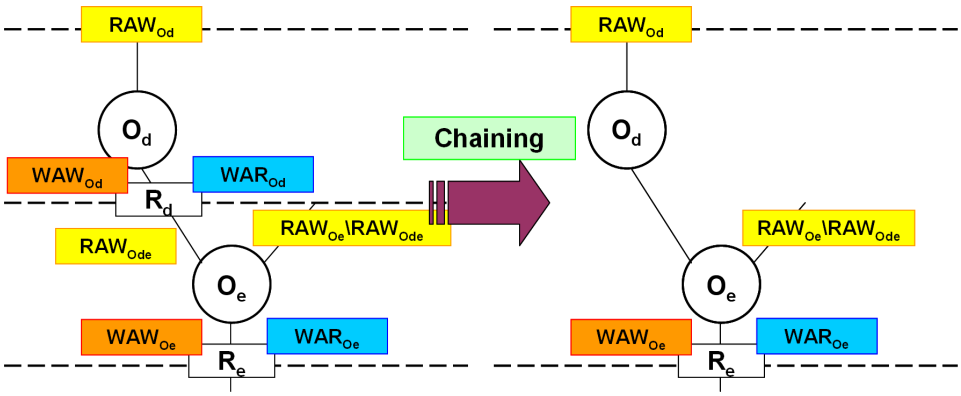


Figure 4.15: Hazards reorganization when considering chaining

in [RSMMH05]. On the contrary, if logarithmic modules are used, the critical path of the adders will be duplicated and therefore the cycle time will increase considerably.

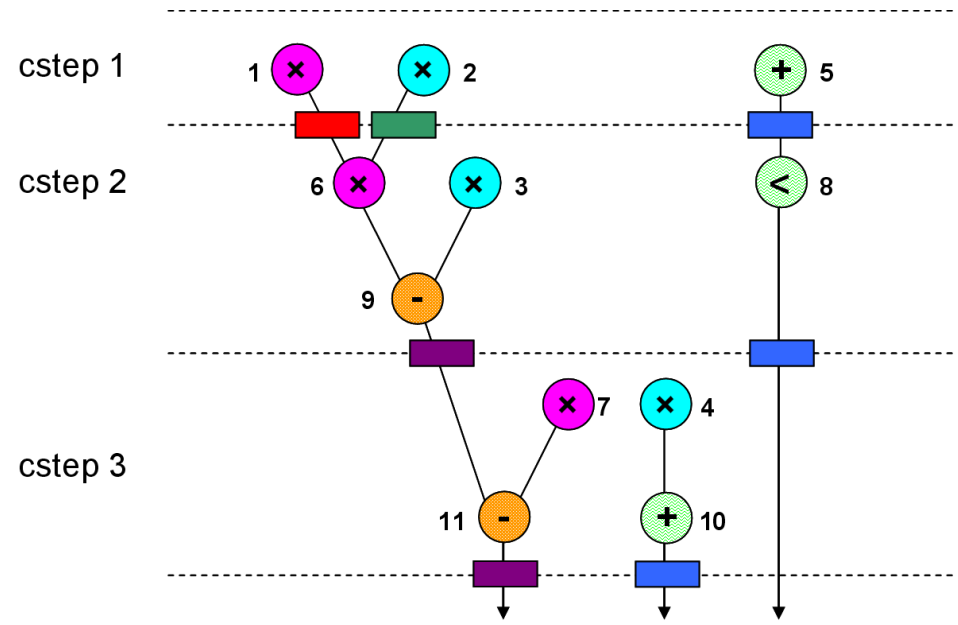
4.3.2.1. Application example

An example about how to apply the aforementioned modifications over the Distributed Management architecture will be described in this subsection.

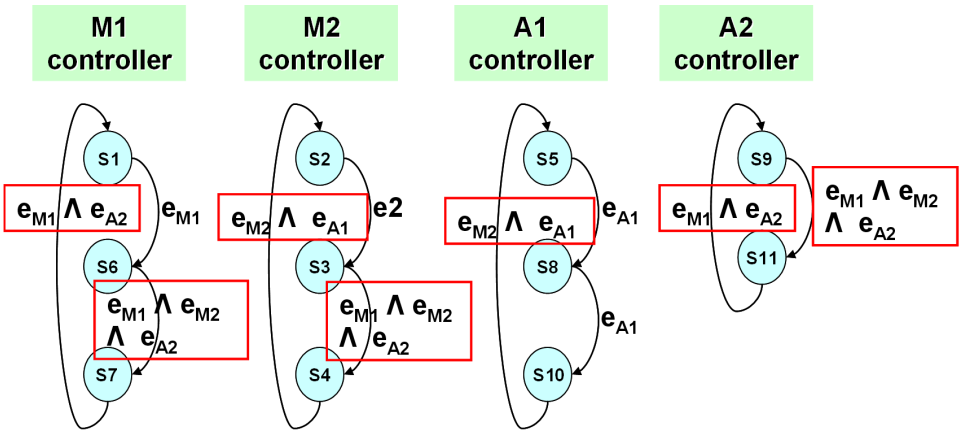
Let's consider the scheduling and binding proposed by figure 4.16a. As explained in the previous subsection, the signals that will fire the state transitions will be those enables corresponding to the SFUs where the chained operations are bound. This can be seen in figure 4.16b. If these new controllers are compared with those of the original example, depicted in figure 4.4, it is observed, for instance, that as *Operations 6*, *3* and *9* are chained, the condition that will fire the transition in M1, M2 and A2 controllers is the logic **AND** of e_{M1}, e_{M2} and e_{A2}. For the rest of the cases the firing signals are obtained in the same way.

The logic for controlling the registers will be diminished, but only because less operations are written. As it can be observed in figure 4.17a, only *Operation 1* will be written in R1, so $load_{R1}$ will only be fired by this operation.

Finally, see the iteration controller, i.e. the T-flipflop, in figure 4.17b. In spite of chaining, the last operation bound to M1, i.e. *Operation 7*, will remain the same, so the glue logic for toggling the T-flipflop will be the same.

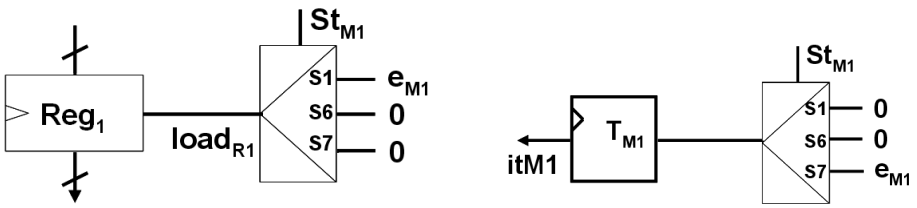


(a) DiffEq scheduling and binding with chained operations



(b) DiffEq local SFU controllers

Figure 4.16: Impact of chaining over the DiffEq SFU controllers



(a) R1 Signals (b) M1 T-flipflop control

Figure 4.17: Impact of chaining over the DiffEq control with the scheduling/binding shown in figure 4.16a

4.4. Example of use

In this section the motivational example shown in figure 1.5 will be re-examined, but considering the foundations of the Distributed Management technique, that have been explained in the previous sections. The local controllers evolution with monocycle and multicycle SFUs will be shown.

4.4.1. Monocycle SFUs

In order to better illustrate how the proposed Distributed Management scheme works with the principles outlined in this chapter, let's reconsider the execution flow of the DiffEq example with monocycle SFUs, as shown in figure 4.18. The additional columns on the right describe the evolution of the local states and the T-values associated with every controller.

In the first cycle there is a misprediction in adder A1, so only its controller is stopped until cycle 2, where the operation is corrected and then written. Controllers of M1 and M2 make a transition to states S6 and S3 respectively, while the A2 controller remains in state S9 because *Operation 9* has not been issued yet. In cycle 3 *Operation 9* is ready, so it is issued, and besides committed, and therefore its controller makes a transition to state S11. Also in cycle 3, M1 and M2 finish the execution of their three bound operations, so they change their T-values and advance to the next iteration. A1 and A2 do the same in cycle 4. In cycle 6, *Operations 7* and *8* suffer a misprediction, so their controllers are stopped and operations are corrected in cycle 7.

Let's compare this with the case of the Centralized Management, which was shown in figure 3.4. Two operations statically scheduled in different cycles could not be corrected in the same cycle with Centralized Management, but with Distributed Management they can, thereby, hiding the penalty cycles.

In cycle 6, M2 finishes its second iteration, so its controller changes the T-value. Hence M2 starts its third iteration in cycle 7, but *Operation 2* remains stalled because it has not resolved its WAR hazard with *Operation 10*. In cycle 7, M1 also finishes its second iteration, so its controller changes its T-value. Finally in cycle 8, *Operations 10* and *11* are committed and therefore A1 and A2 finish their second iterations, changing their T-values for the next cycle.

At this point DiffEq has completed two iterations. Assuming that cycle time in the speculative case is roughly 75 % of the non-speculative one, as in the analysis performed in section 3.4.1, the execution time will be 8 cycles * 0.75 *time units*/cycle = 6 *time units*. Furthermore, note that *Operations 1* and *2* from the third iteration have been written too, so the exact performance would be even higher.

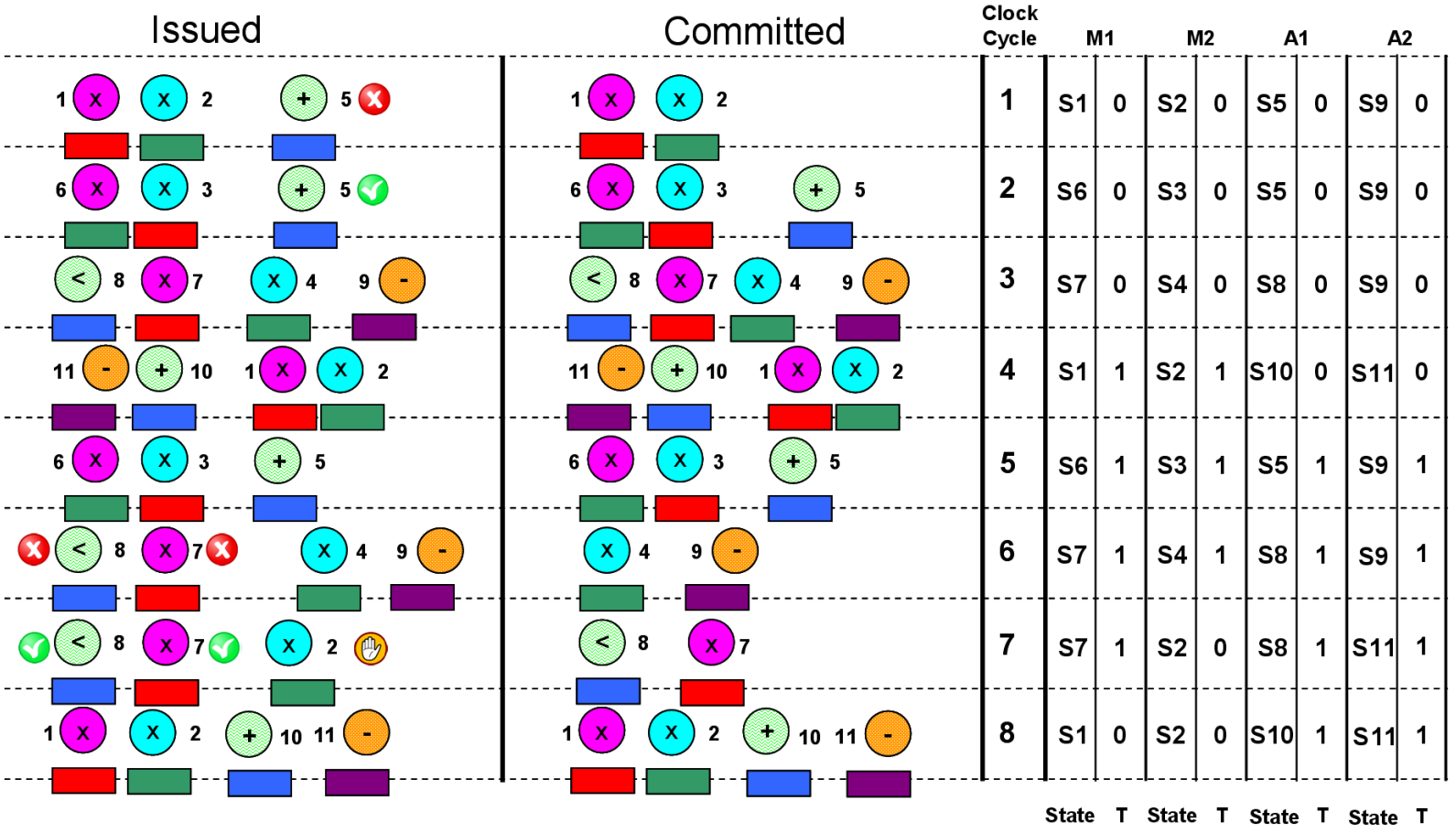


Figure 4.18: DiffEq execution flow with monocycle SFUs and Distributed Management

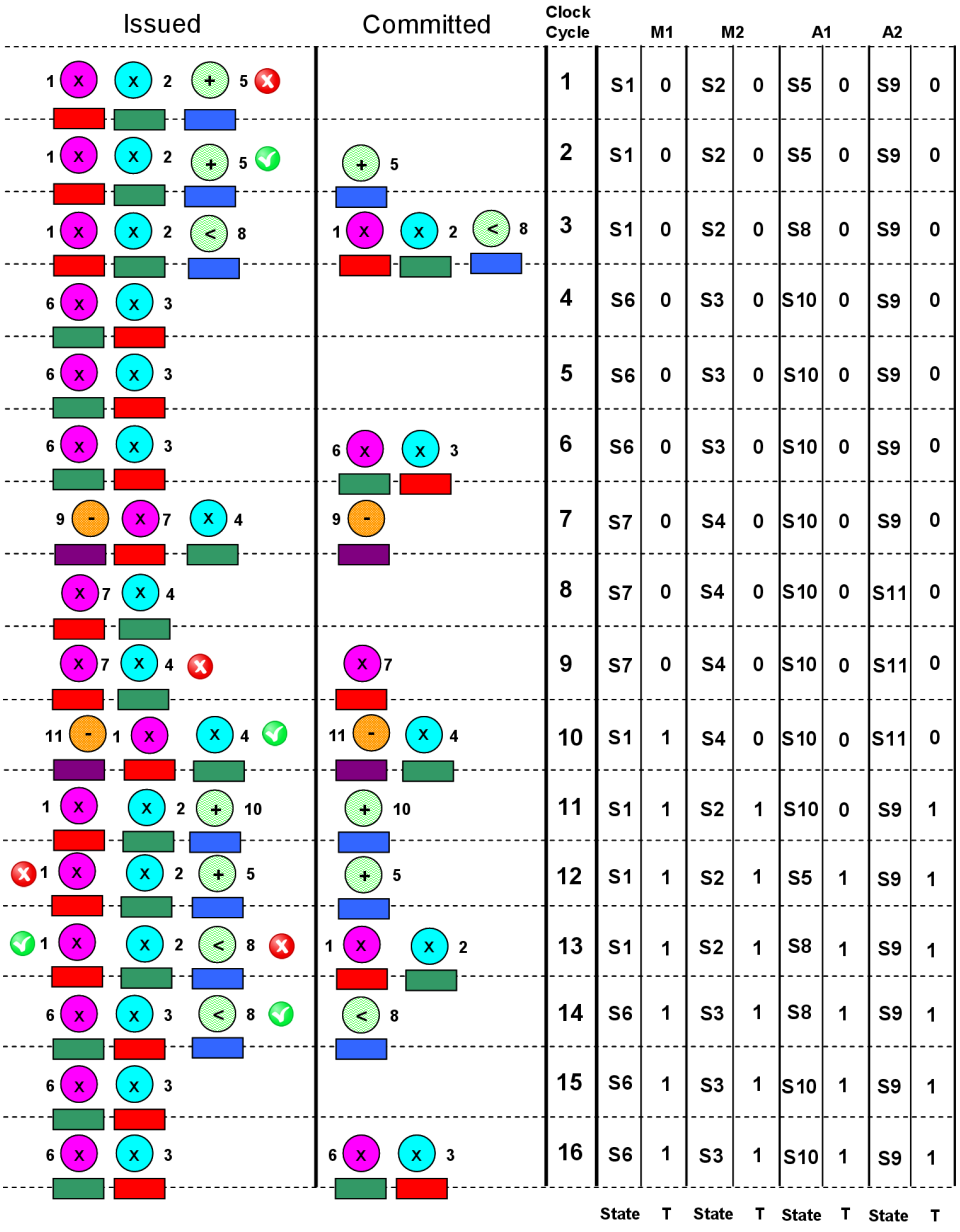


Figure 4.19: DiffEq execution flow with multicycle SFUs and Distributed Management

4.4.2. Multicycle SFUs

As in the case of Centralized Management, the introduction of multicycle SFUs will be evaluated with the execution flow of the DiffEq example. However, there is no need to consider a static scheduling with multicycle FUs,

as the one shown in figure 3.5, because the Distributed Management architecture will take into account the FUs latencies without adding more states. Moreover the reader must consider the timing model described in subsection 2.3, i.e. the best case latency is 3 cycles for the multipliers and 1 cycle for the adders, plus one more cycle if there is a misprediction.

At first glance it is easy to see that with multicycle SFUs there will be less committed operations per cycle, because they will only be committed when the SFU where they have been bound to has completed the corresponding FU-latency. Secondly, it is interesting to note that additions, and low-latency operations in general, will hide their penalty cycles very easily while long-latency operations are being performed. This was not the case of Centralized Management, where one stall cycle was introduced every time there was a misprediction in whatever the operation.

Then let's examine figure 4.19 step by step. In the first cycle *Operation 5* suffers a misprediction, but it is corrected while *Operations 1* and *2* are being executed and, therefore this penalty cycle is hidden. The only consequence is that A1 controller remains in state S5 one more cycle. Afterwards in cycle 3, *Operation 8* is executed in adder A1 and the execution flow continues as if no misprediction had happened before.

In cycle 9 *Operation 4* has a misprediction and only *Operation 7* can be committed. Hence the M1 controller makes a transition to state S1, and in this case the T-value is flipped, while the M2 controller remains stalled in state S4, and maintaining the T-value of the current iteration. In cycle 12 *Operation 1* suffers a misprediction and therefore it will need an extra cycle to be corrected. Thus the M1 controller will remain in S1 one more cycle. Finally a failure happens in cycle 13 in *Operation 8*, so the A1 controller remains in S8 one more cycle, and therefore *Operation 8* will be corrected in cycle 14 without stalling the execution of other operations.

Overall, if no more failures are supposed, two iterations of the DiffEq benchmark will be completed in 20 cycles, which is 16.7 % better in comparison with the 24 cycles of the Centralized Management implementation, and 28.6 % better with respect to a conventional implementation. Note that cycle length will be nearly the same in the case of multicycle non-speculative and speculative FUs, the only difference is the number of cycles they take to complete an operation; so comparing the latency of the circuit will be enough for giving an idea about execution time.

4.5. Improving performance via HLS techniques

Traditional techniques have been established for traditional implementations. However the dynamic scheduling techniques described in this Ph.D. Thesis introduce a new execution paradigm which can be further exploited if some refined techniques are used in the HLS process. In concrete the mod-

ification of allocation and binding will be analysed and its impact on the speculative datapaths will be studied.

4.5.1. Allocation

Allocation is the task of assigning operations onto available Functional Units types. Hence, different FUs could have a different impact on the Distributed Management architecture. The FUs utilized in the experiments of the previous chapters, except in subsection 3.5.2.3, follow a linear implementation, such as the Ripple Carry Adders or the Baugh-Wooley Multipliers. Thereby, different FUs, or at least Functional Units with different implementation styles, should be tested. That's why logarithmic-like modules will be considered too in the experiments section. These modules will follow the specifications explained in section 2.3.

4.5.2. Binding

Binding consists in the task of assigning a set of resources to an operation, i.e. FUs, registers, multiplexers, etc. On the one hand, performance is affected by the number of failures, i.e. the predictors, which are located inside the FUs and are highly influenced by data correlation. Thus their performance depends on the FU inputs.

On the other hand, overall performance is determined by the RAW, WAR and WAW hazards. As RAW hazards are inherent to the initial specification, WAR and WAW hazards must be diminished as much as possible to increase performance. These two hazards are dependencies produced by the reading and writing of the same register.

Therefore in this subsection, FU and register binding will be evaluated in order to better fit to the Speculative Execution Paradigm.

4.5.2.1. FU Binding

As it has been explained in section 2.1.6, the adders considered in the experiments sections of this Ph.D. Thesis utilize predictors in order to provide the carry-in to the most significant fragment of the additions that they are executing. These predictors are based on data correlation. Besides many of them use patterns composed by some operands' bits to access to the proper prediction.

In general, similar operands will produce similar carry chains, so as similar operands access to the same prediction, they will produce more hits. Hence, increasing data correlation is a good way to improve predictors accuracy.

In order to increase correlation between consecutive operations bound to the same FU, the *Hamming FU* (HFU) binding has been proposed to reduce

the *Hamming Distance* (HD) between the profiled most common patterns of every pair of consecutive operations. For more information about the most common patterns the reader is invited to look up appendix A.

The pseudocode of the HFU binding is shown by algorithm 4.4. It receives a list of operations, a list of unbound Functional Units and a given latency λ, and it returns the list of bound Functional Units. Note that the list of unbound FUs is a design constraint, i.e. the number and type of FUs. In the algorithm, every operation of every cstep is bound to the FU where the Hamming Distance with respect to its last bound operation is miminum. This Hamming Distance is computed between the patterns of the operations. Commutative property between the operands is also evaluated in order to diminish Hamming Distance as much as possible.

This algorithm is greedy, so its complexity is polinomial. Let n and f be the number of operations of the DFG and the number of FUs, respectively. The method *getOperationsInCstep* is $\mathbf{O(n)}$, because it searches among all the operations those that are scheduled in a concrete cstep. The method *hammingDistanceBindOperation* looks for the best FU according to the HD heuristic; so its complexity is $\mathbf{O(f)}$. Finally, the *updateBusyFUs* method updates the life of the variables bound to every FU; so it is $\mathbf{O(f)}$. Clearly the algorithm complexity will be dominated by the inner loop. The inner loop will be executed n/λ times on average each iteration. Hence its complexity is $\mathbf{O(nf/\lambda)}$. Therefore the overall complexity of the algorithm will be $\mathbf{O(\lambda nf/\lambda) = O(nf)}$.

In the same way, the reader may think to increase the correlation via the static scheduling. However it is not a practical solution. The idea would consist in scheduling operations with similar input patterns in consecutive cycles for increasing data correlation. Note that these input patterns can be profiled before. However this could lead to suboptimal solutions. On the one hand some operations in the critical path could be delayed and thus latency increased. A solution could be the use of a heuristic which combines

Algorithm 4.4 hammingFUBinding(operations, unboundFUs, λ)

```
1: boundFUs ← unboundFUs
2: for i = 1 to λ do
3:    operationsInCstep ← getOperationsInCstep(operations, i)
4:    for j = 1 to #(operationsInCstep) do
5:       operation ← operationsInCstep.get(j)
6:       hammingDistanceBindOperation(operation, boundFUs)
7:    end for
8:    updateBusyFUs(boundFUs)
9: end for
10: return boundFUs
```

both mobility and Hamming Distance for complying with the initial specification while increasing correlation. However if the latency is adjusted for achieving a maximum performance, in the initial specifications there will be few operations with large mobilities. In this scenario, every cstep would be determined by the mobility rather than the Hamming Distance, and a priority list-based scheduling or a traditional *Force Directed Scheduling* [Mic94] would be enough.

4.5.2.2. Register Binding

Since registers are critical in every dynamic scheduling scheme, registers binding becomes an important task. With Distributed Management, penalty is produced from failures in the prediction, but there are also situations where some operations must wait for their source registers. Two alternatives have been studied in order to diminish the number of stalls, namely:

(1) Reducing Hamming Distance between consecutive operations bound to the same register. The *Hamming Register* (HR) binding has the same foundation as HFU, since results that must be written can be also profiled. If similar results are written in the same register, they will produce similar carries when used as source registers. Nevertheless, if every register feeds different FUs, the effect of this binding will be highly reduced. But, if the FU binding is such that the operands that feed a concrete FU are similar, the impact of HR will be higher. Hence, combining HR with HFU is a good option for increasing correlation between operands and thus, the efficiency of HR.

(2) A *Least Recently Used Register* (LRUR) binding policy. This alternative tries to separate as much as possible the writing of the same register. Therefore binding different registers to close operations in time will help to diminish the number of dependencies and thus penalty will be reduced. It must be noticed that only WAR and WAW hazards can be removed with this technique. Evidently, the RAW hazards cannot be eliminated with a different register binding, because these hazards are inherent to the DFG, not to the registers.

A generic pseudocode of the customized register bindings is shown by algorithm 4.5. Independently of utilizing HR or LRUR, the structure of the algorithm is the same. It receives the operations and unbound registers lists, and a given latency λ, and it returns the list of bound registers. The initial unbound registers list is calculated with the same binding as the one considered in the conventional non-speculative case. In the experiments of this Ph.D. Thesis, a *Left-Edge Algorithm* (LEA) [Mic94] has been utilized. This initial registers list is given in order to maximize the reduction in terms of the applied heuristic. Intuitively, the more available registers in every cstep,

the more probabilities to find a register with smaller Hamming Distance or less Recently Used.

In algorithm 4.5 every operation of every cstep is bound to the most suitable register according to the chosen heuristic (Hamming Register or Least Recently Used Register). In the end of the outer loop, the life of the variables bound to every register will be updated, i.e. in every cstep.

The complexity analysis of algorithm 4.5 is similar to the one performed for algorithm 4.4. If n and r are the number of operations and registers, respectively, the complexity will be $\mathbf{O(nr)}$. The algorithm complexity will also be dominated by the inner loop, which is executed n/λ iterations on average. The complexity of the *customizedBindOperation* method is $\mathbf{O(r)}$, because it searches the best candidate among the r registers. Therefore, the algorithm 4.5 complexity is $\mathbf{O(\lambda nr/\lambda)} = \mathbf{O(nr)}$.

In order to see how a bad register binding impacts over performance, see figure 4.20 and table 4.4. They depict an alternative register binding proposed for the DiffEq example. Concretely, *Operation 9* is bound to register R3, instead of R4 as in the original example, which was shown in figure 4.1.

Now let's examine how the execution flow would be with this alternative binding (see figure 4.21). The same failures than in figure 4.18 have been supposed. As it can be observed, several stalled operations have appeared in the *Issued* column. The final result is that, in comparison with the original binding proposed in figure 4.1, two more cycles are necessary.

The reason is quite simple: by binding *Operation 9* to register R3, two WAW hazards with *Operations 8* and *10*, and a WAR hazard with *Operation 4* have appeared, increasing the risk of stalled operations in the datapath and thus, degrading performance. Ten cycles are required for completing 2 iterations. Considering the timing analysis performed in section 2.3 an execution time of 10*0.75=7.5 *time units* will be necessary for completing two iterations of the DiffEq example. This is worse than the 6 *time units* of the original DiffEq binding with Distributed Management, but still better

Algorithm 4.5 customizedRegBinding(operations, unboundRegisters, λ)

```
1: boundRegisters ← unboundRegisters
2: for i = 1 to λ do
3:    operationsInCstep ← getOperationsInCstep(operations, i)
4:    for j = 1 to #(operationsInCstep) do
5:       operation ← operationsInCstep.get(j)
6:       customizedBindOperation(operation, boundRegisters)
7:    end for
8:    updateBusyRegisters(boundRegisters)
9: end for
10: return boundRegisters
```

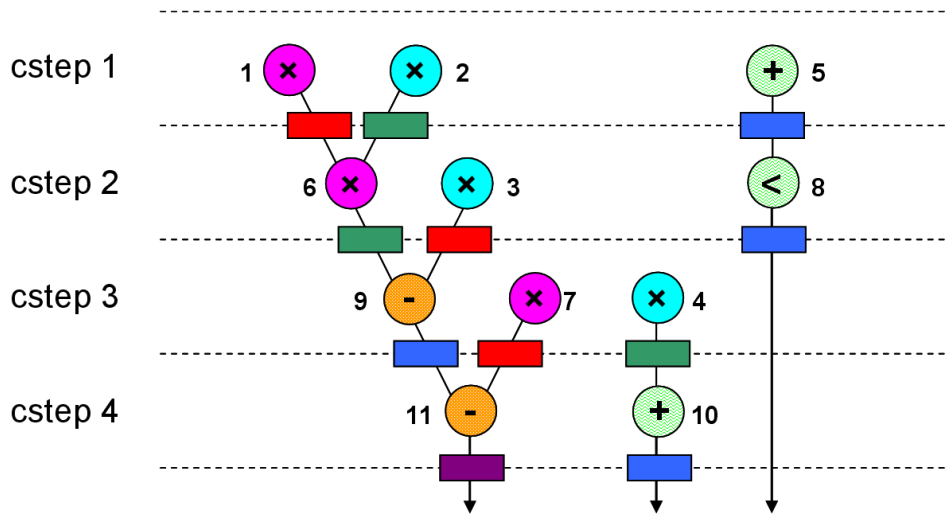


Figure 4.20: Alternative DiffEq scheduling and binding

cstep	M1	M2	A1	A2	R1	R2	R3	R4
1	1	2	5		1	2	5	
2	6	3	8		3	6	8	
3	7	4		9	7	4	9	
4			10	11			10	11

Table 4.4: DiffEq alternative binding summary

than the 8 and 8.25 *time units* resulting from the use of conventional and Centralized Management implementations, respectively.

4.6. Experimental Results

The efficiency of the Distributed Management technique will be experimentally demonstrated in this section. Execution time and area results will be compared with conventional implementations and with the Centralized Management results. As in chapter 3, there will be two conventional implementations, namely: one using Ripple Carry Functional Units and one using Carry Select modules.

4.6.1. Framework

As the experimental framework is the same as with the Centralized Management, no subsection will aim to explain it deeply in this chapter. Hence, for more information about settings the reader is invited to check section

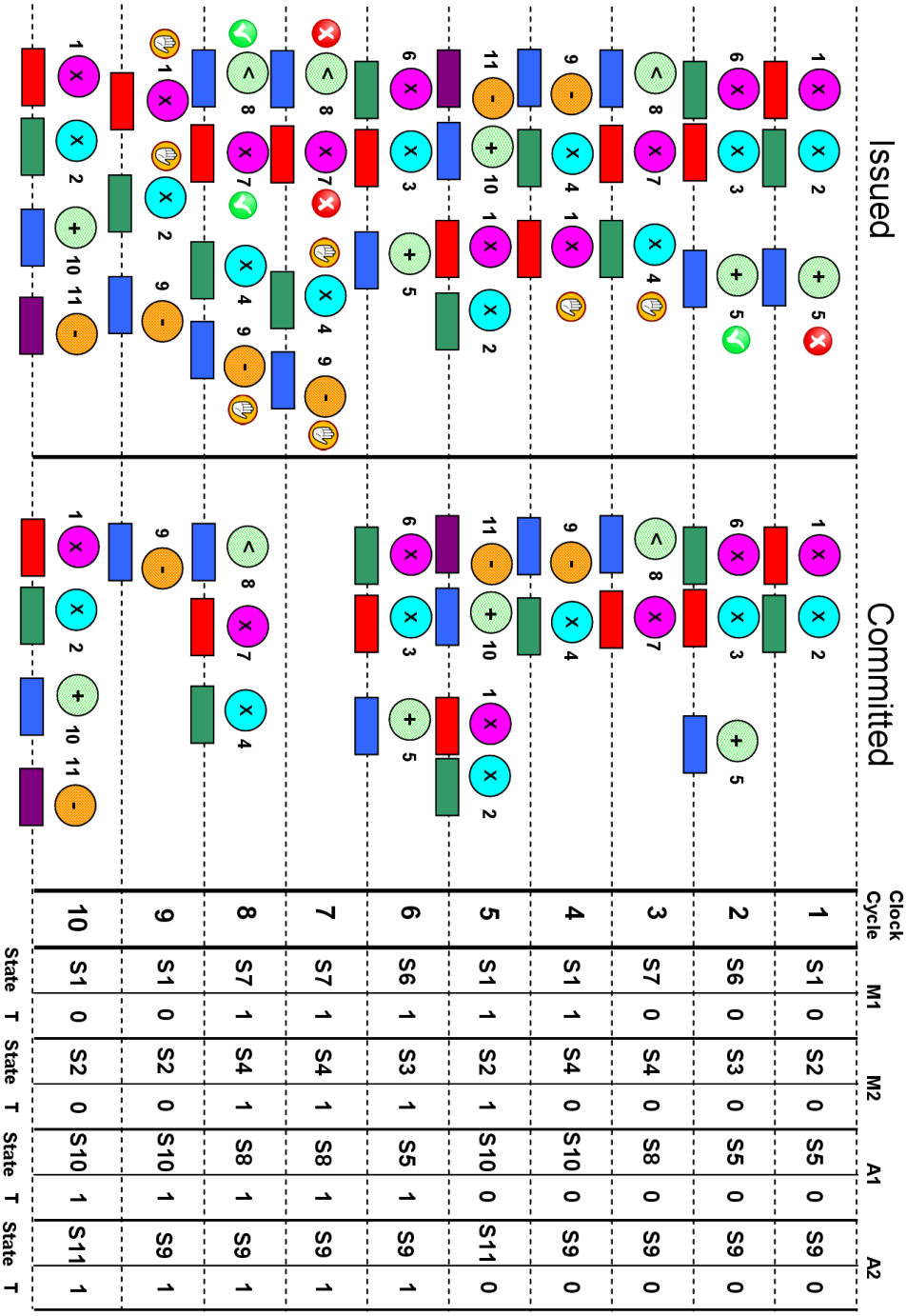


Figure 4.21: DiffEq execution flow with Distributed Management, and based on the alternative binding

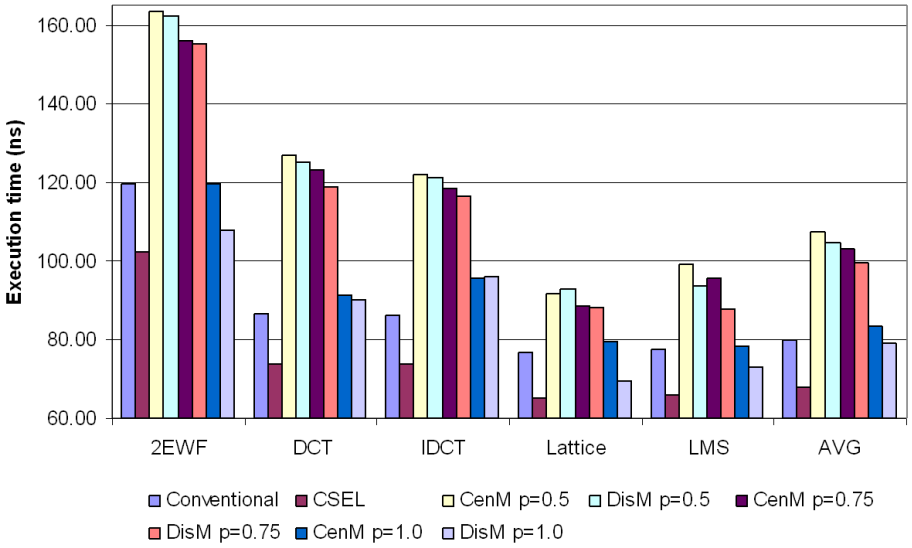


Figure 4.22: Execution time with monocycle SFUs

3.5.1.

4.6.2. Synthesis results

The same six benchmarks that were simulated and synthesized with the Centralized Management technique, have been simulated and synthesized with the Distributed Management one, and with monocycle and multicycle SFUs. Furthermore, these new results will be compared with those obtained in the previous chapter with the conventional implementation with RCA-like modules, the CSEL and the Centralized Management implementations.

Figures 4.22, 4.23 and 4.24 offer a summary of the execution time of the six benchmarks and the average case with different implementation styles, namely: conventional, conventional with Carry Select Adders (CSEL), Centralized Management (CenM) and Distributed Management (DisM). Besides in the speculative cases, three values of the p parameter have been tested.

First a study with monocycle FUs has been performed. Execution time results are shown in figure 4.22 and the cycle time ones in table 4.5. Distributed Management results are still worse than using a conventional implementation. In concrete, with p=0.5 DisM increases 30.8 % execution time, while with p=1.0 execution time is reduced 0.65 %. This improves Centralized Management results, but it is not enough yet, since CSEL average execution time is 16.7 % less than Distributed Management in the ideal case, i.e. p=1.

Benchmark	Conventional	CSEL	CenM	DisM
DiffEq	8.35	7.09	6.74	7.09
2EWF	8.56	7.31	7.46	7.86
DCT	8.67	7.38	7.31	8.04
IDCT	8.64	7.38	6.92	7.86
Lattice	8.54	7.23	6.93	7.23
LMS	8.63	7.34	7.11	7.58
AVG	8.56	7.29	7.08	7.61

Table 4.5: Cycle time summary using monocycle FUs

Distributed Management cycle time is lower than in the conventional implementation, but higher than with CSEL or Centralized Management. This is due to the higher complexity of the Distributed Management control. Thereby, although there are less stalls than with Centralized Management, which implies a latency reduction, it still penalizes too much overall performance.

Now let's examine deeply cycle time results, shown in table 4.5. As it has been said, Distributed Management delay overhead is greater than with Centralized Management. Hence, the theoretical 21 % cycle time gain, because of the speculative multiplier, is not fulfilled. On average cycle time is reduced 11.1 % with respect to the conventional implementation, which is less than the 17.3 % reduction obtained with the Centralized Management. However this loss is compensated with the latency reduction and overall performance improvement. Furthermore, as in the Centralized Management case, delay and area penalties will become less significant as data widths are increased.

4.6.2.1. Multicycle FUs and chaining impact

Figure 4.23 depicts the execution time results with the same implementation styles than in figure 4.22, but utilizing multicycle SFUs. As in the case of Centralized Management, the corresponding cycle time has been obtained by dividing the monocycle cycle time by the multiplier latency in the hit case. See section 3.5.2 for more details.

The introduction of multicycle SFUs boosts Distributed Management performance. In concrete, Distributed Management reduces 22.9 % and 8.5 % execution time with respect to the conventional and the CenM implementations. Besides, execution time is nearly the same than with CSEL. Note that this is achieved in the worst case, with no correlation at all, i.e. p=0.5. With greater correlation levels, Distributed Management can reduce execution time up to 28.3 %, 6.1 % and 7.2 % with respect to the conventional, CenM and CSEL implementations. Nevertheless, as these reductions are obtained with the ideal case, i.e. p=1, actual improvements with respect to the

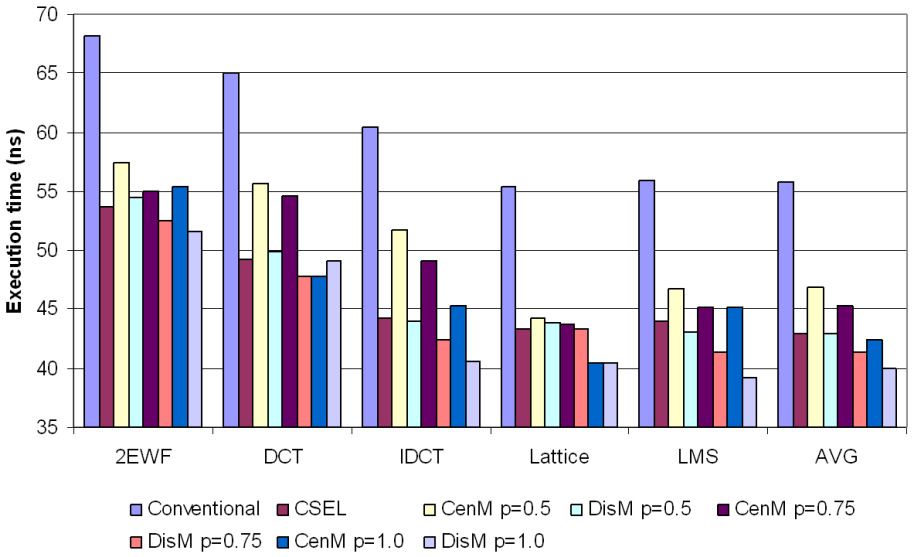


Figure 4.23: Execution time with multicycle SFUs

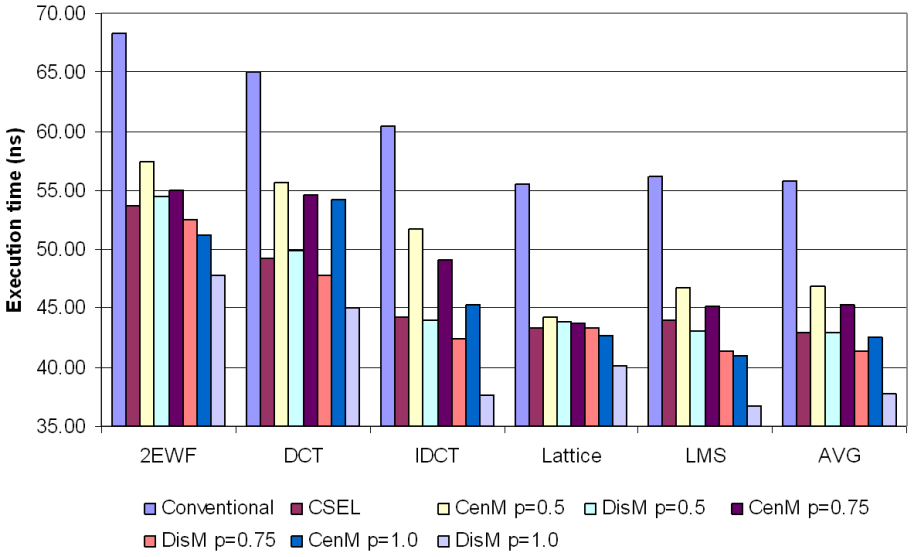


Figure 4.24: Execution time with multicycle SFUs and chaining

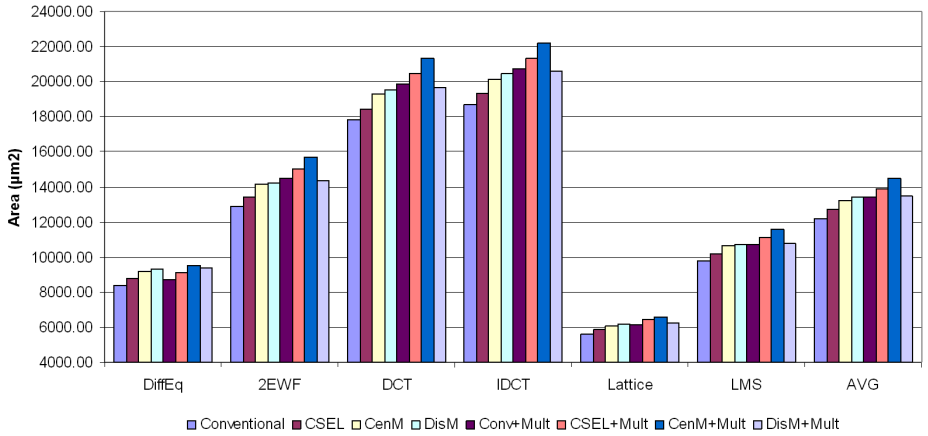


Figure 4.25: Area comparison with different implementation styles

non-speculative implementations are slightly lower.

The impact of chaining in the Distributed Management performance is something better than in the Centralized Management case, and it can be observed in figure 4.24. Considering chaining, Distributed Management diminishes 23 % and 8.4 % execution time with respect to a conventional and a CenM implementation, and it matches CSEL performance. This is achieved in the worst case, but with p>0.5 Distributed Management can reduce execution time up to 32.5 %, 11.5 % and 12.4 % with respect to the conventional, CenM and CSEL implementations, respectively.

Therefore, it can be concluded that chaining impact over Distributed Management is around 4 % execution time reduction. Removing some dependencies, by avoiding the writting of some registers via chaining, is a better scenario for Distributed Management. Operations that originally depended on the removed registers, may begin its execution before than without chaining. In the case of Centralized Management, or with a conventional implementation, this is not possible because operations must always be synchronized according to the static scheduling. Nevertheless, as the latency and FU constraints are really stringent, only few additions can be chained and thus performance is not much better than without chaining.

4.6.2.2. Area penalty analysis

Area results are shown in figure 4.25. As it can be observed, area penalties are really low. First let's examine the implementations with monocycle FUs, which correspond with the four leftmost bars inside every set of them. CSEL, CenM and DisM occupies 4 %, 8.8 % and 10.2 % more than the conventional case. In other words, the area penalty due to the Distributed Management

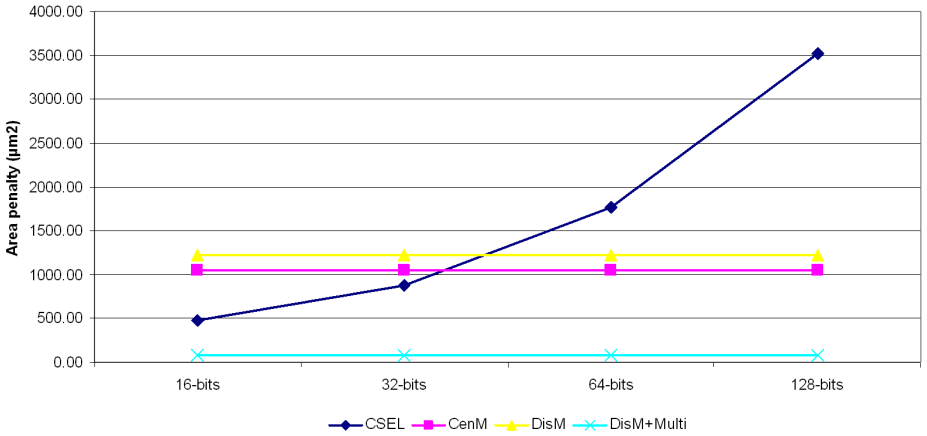


Figure 4.26: Area penalty with respect to data width

is only 1.4 % higher than with Centralized Management, in spite that DisM seems to be much more complex than CenM. Nevertheless, as only CSEL clearly improves the baseline performance, it is the only interesting option. However, CSEL area penalty depends on data width, as it was explained in section 3.5.2, so it must be analyzed carefully.

The four rightmost bars of every column depict the area penalty when considering multicycle FUs and chaining. Results show that CSEL, CenM and DisM produce an area penalty of 3.6 %, 7.8 % and 0.6 % with respect to the multicycle baseline case. The Distributed Management area overhead has been reduced 9.6 % when considering multicycle FUs. As it has been described in this chapter, the inclusion of multicycle FUs in the Distributed Management architecture only needs a counter and some glue logic, while in a conventional or Centralized implementation the number of required states increases rapidly and thus the area needed by the global controller. Therefore, when considering multicycle FUs and chaining, both area and execution time results are better with Distributed Management than with CSEL.

A detailed study of the area penalty with respect to data width is depicted in figure 4.26. Area penalties in the case of CSEL, CenM, and Distributed Management with monocycle SFUs (DisM) and with multicycle and chaining (DisM+Multi) are depicted. First, the reader must take into account that these results are relative to the corresponding baseline case, i.e. the baseline case with monocycle SFUs is not the same than with multicycle SFUs and chaining. Second, as it has been said in section 3.5.2, the area overhead of CSEL and CenM is the same with monocycle and with multicycle SFUs. Besides, note that chaining impact over area is negligible, because it does not imply a significant change in the number of states, specially if multicycle

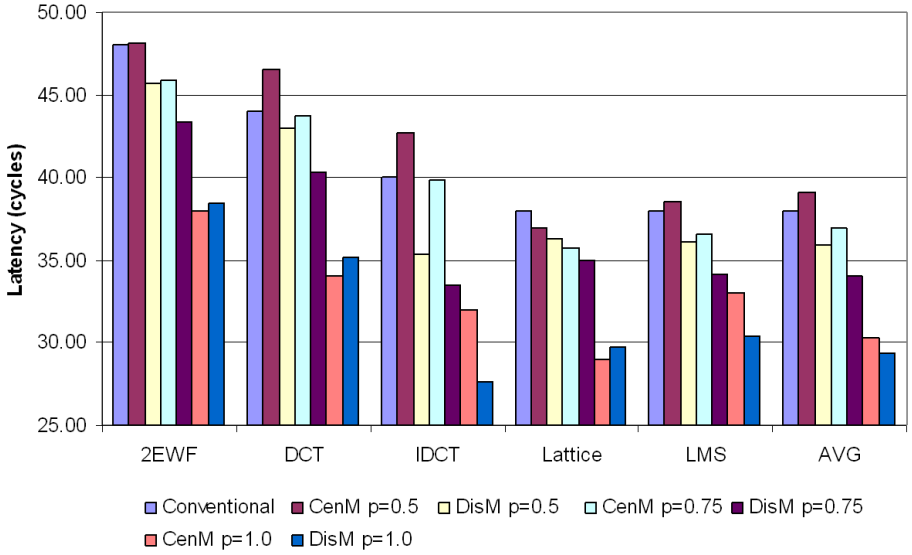


Figure 4.27: Latency with logarithmic FUs and Distributed Management

FUs are being used.

Centralized Management and Distributed Management area overheads remain constant with respect to data width increase. On the other hand, as CSEL penalty depends on data width, CSEL implementations will produce a higher area overhead as this parameter increases. Finally, it is interesting to note that the Distributed Management with multicycle SFUs area penalty is almost zero, and much less than in the case of monocycle SFUs. The reason is that with multicycle FUs the conventional controller increases quite a lot in comparison to the augmentation produced in the Distributed Management, because of the very little additional required logic.

4.6.2.3. Using logarithmic FUs

The inclusion of speculative logarithmic modules has also been tested with the Distributed Management technique. The settings of this experiment only include multicycle SFUs and the logarithmic modules and timing model from section 2.3. Chaining is not considered since it would increase the resulting cycle time, as explained in section 4.3.2.

Latency results are depicted by figure 4.27. As in the case of Centralized Management, the implementation of these modules is not available, so only latency results can be given. The reader must remember that logarithmic FUs presented in [VBI08] reduce the adder delay, which will diminish cycle time. Hence, a latency reduction will be translated into an execution time de-

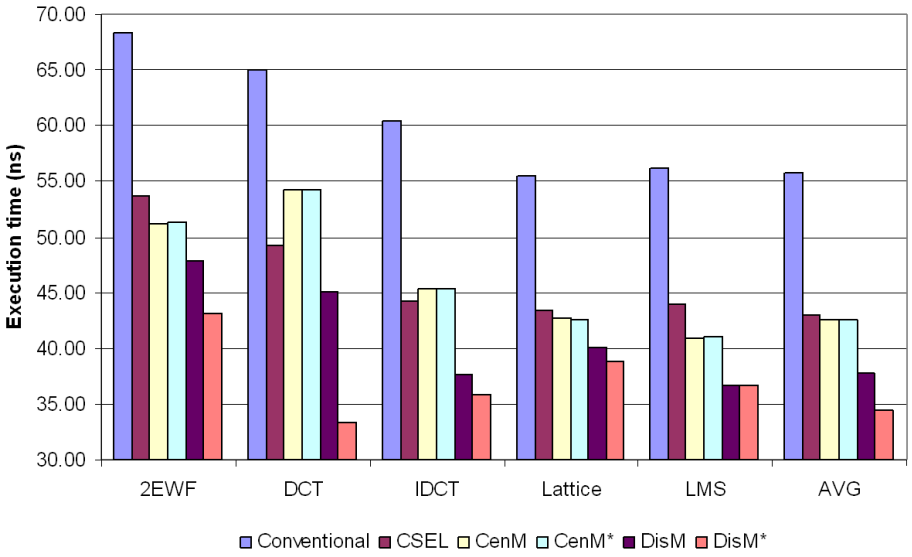


Figure 4.28: Impact of customized binding

crease. Distributed Management improves Centralized Management results, specially when correlation is low. DisM diminishes 5.4 % latency with p=0.5, 10.4 % with p=0.75, 7.6 % more than CenM, and it can achieve a maximum 22.6 % latency reduction in the ideal case of correlation, 2.5 % more than CenM.

These latency reductions may not seem very significant. However the reader must remember that the multicycle controller penalty will probably cause CenM implementation to occupy more area than the DisM one. Moreover, some improvements will be described in the following experiment, which will increase the difference between Distributed and Centralized Management execution times without incurring a higher area penalty.

4.6.2.4. Impact of customized binding

The six benchmarks have been simulated with the bindings explained in subsection 4.5.2 and with multicycle FUs and chaining. Results are shown in figure 4.28. As the objective of this experiment is to measure the difference between the non-customized and customized binding, only results corresponding with p=1 are depicted. This difference between customized and non-customized binding remains quite similar accross the whole range of p. Note that the options anotated with an * are the ones with customized binding. Furthermore, the conventional and CSEL implementations are depicted.

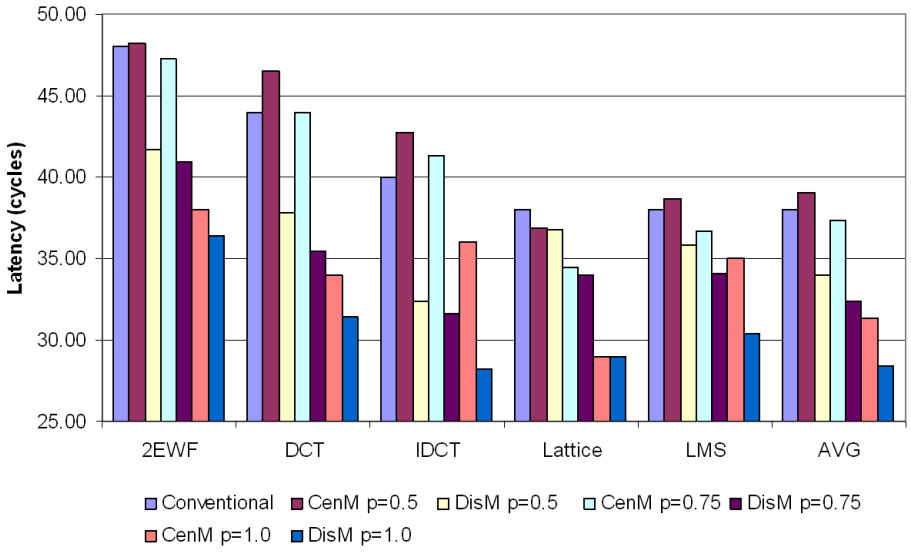


Figure 4.29: Impact of customized binding with logarithmic FUs

Experiments show that customized binding has no effect over the CenM option, because results for both CenM and CenM* are practically identical. However, in the case of DisM*, customized binding reduces execution time by 8.6 % on average with respect to the naïve binding techniques applied to DisM and by 19 % and 38 % on average with respect to CenM and the conventional implementation with no speculation.

In 5 of the 6 considered benchmarks, the best combination of FU-binding and register binding is Hamming FU binding with a Least Recently Used Register binding. In other words, increasing data correlation, i.e. reducing Hamming Distance, between the operations bound to the same FU is better than a naive FU binding. From the registers' point of view, separating the use of the same register as much as possible produces better results than diminishing the Hamming Distance of the results written in the registers.

More results can be seen in figure 4.29. The same settings as in the previous experiment with logarithmic SFUs have been supposed, i.e. with no chaining. As it can be observed, customizing the binding with HFU and LRUR has no impact over CenM, but it achieves an extra 4-5 % latency reduction when considering DisM, reaching 25.3 % peak average reduction, i.e. p=1, with respect to the conventional implementation without speculation.

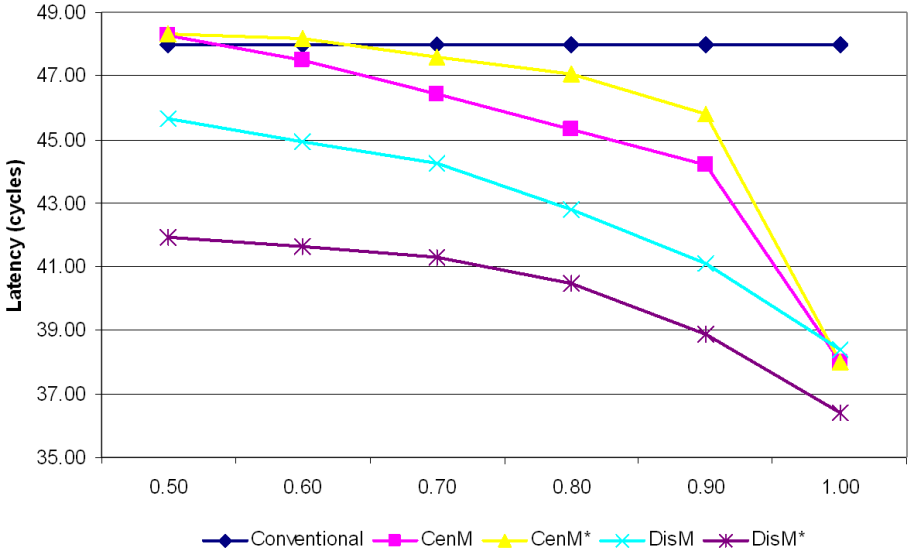


Figure 4.30: Evolution of latency with respect to p, for the 2EWF benchmark with logarithmic FUs

4.6.2.5. Sensitivity of latency to parameter p

It is clear that as we increase the value of parameter p, results become better because data correlation increases too. However, this improvement in performance is not the same as the value of p is varied from 0.5 to 0.75 or from 0.75 to 1.0. Therefore, a closer study of the relationship between latency and p has been performed with the 2EWF benchmark. Logarithmic modules have been assumed in this study. Besides, naive and customized binding options have been considered for both Centralized and Distributed Management.

Results depicted in figure 4.30 indicate that for every technique the latency decreases linearly up to p=0.9. From 0.9 to 1.0 the slope is something steeper. So there is a critical point around p=0.9 from where mispredictions diminish more drastically. It should be noted that in the DisM implementations this slope is smoother; so Distributed Management behaves much better through the whole range of parameter p. The final consequence is that Distributed Management works much better than Centralized Management with low values of p.

As real inputs are not available for these benchmarks, synthetic patterns have been used for the primary inputs. However, in several studies performed with the JPEG2000 [Cha99] decoder and with real photos, it is observed that data behave similar to the captured pattern around 85 %-90 % of the cases,

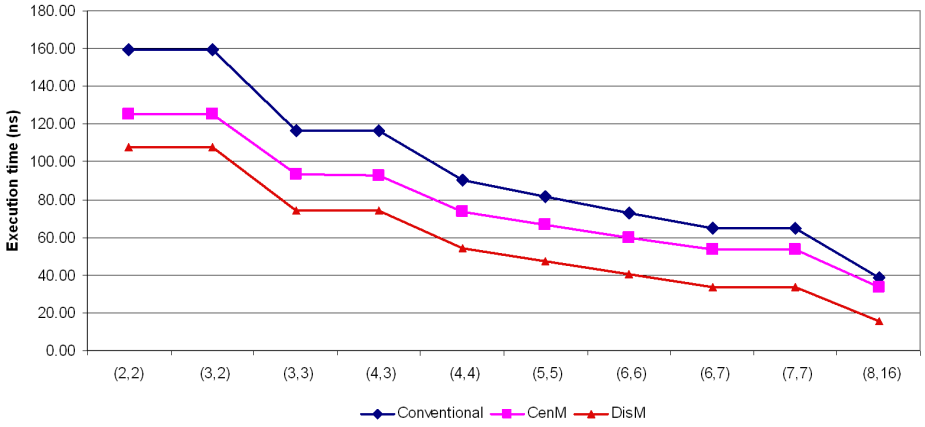
so it can be considered that p=0.85-0.9, although this value should vary depending on the benchmarks.

4.6.2.6. Sensitivity of latency to the number of FUs

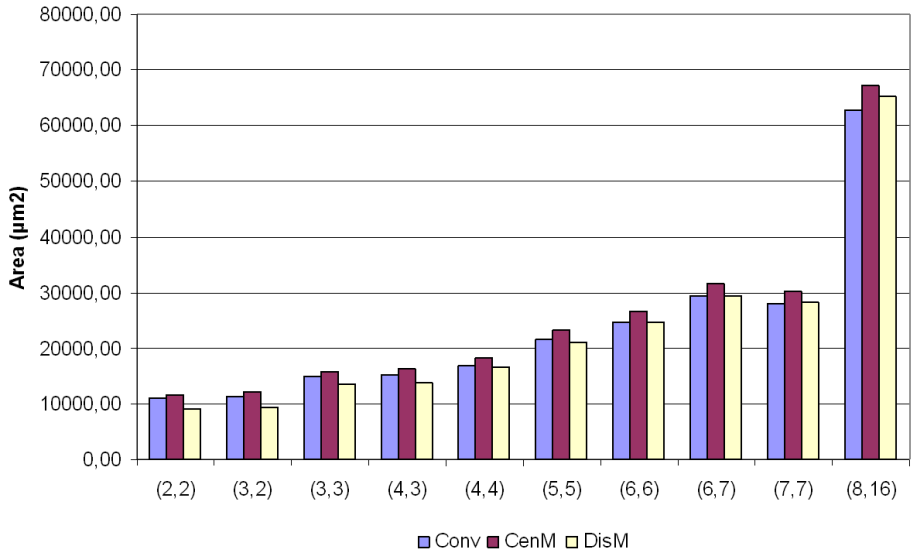
In order to test the execution time benefit and the area overhead produced by the local SFU controllers with respect to the number of resources, the *dot product* of two vectors of 32 components has been implemented. Every element of the vector is represented with 16-bits. Figures 4.31a and 4.31b depict the execution time and area, respectively, with a baseline implementation with non-speculative FUs, labeled as Conventional, with SFUs and Centralized Management, labeled as CenM, and with SFUs and Distributed Management, labeled as DisM. The number of (Adders, Multipliers) is shown inside parentheses below the X axis. In this experiment p=0.85, and multicycle linear SFUs and chaining have been considered. Note that the SFUs latency is given by table 3.2b.

As it can be observed, a reduction in execution time is still achieved in spite of increasing the number of SFUs, i.e. the number of predictors. In other words, the hit rates are good and when there is a failure, the penalty cycles are properly hidden in most of the cases. On average, Centralized Management can obtain 19.4 % execution time reduction, while Distributed Management can reach 39 % reduction.

It may seem that controllers' complexity could increase very much with the augmentation in the number of FUs. However, the SFU local controllers are quite simple, and when considering multicycle FUs they only need to add a counter and some glue logic, while conventional controllers increase too much when the circuit latency is really high. Hence, when using few resources the circuit latency is high and then DisM area is even less than in the conventional case. On the other hand, when the number of resources is increased, the circuit latency is diminished and thus traditional controllers become smaller. On average, CenM causes 7 % area overhead while DisM area is reduced 9 % with respect to the conventional implementation. Hence, the SFUs area scales well.



(a) Execution time



(b) Area

Figure 4.31: Evolution of execution time and area in the Dot product example with different number of FUs

Chapter 5

Conclusions

Art is never finished, only abandoned
Leonardo da Vinci

Dear reader, this is the end of my Ph.D. Thesis. In this chapter, the final remarks of the work will be given. This Ph.D. Thesis has been a hard task. However all the work has not been done yet, and future lines of promising research will be presented as well.

5.1. The final remarks

Historically in High-Level Synthesis everything has been static. In order to optimize ASICs as much as possible, a great deal of pre-synthesis work is done before synthesizing the circuit. Everything can be controlled statically because several constraints are imposed. These constraints can restrict the possibilities of High-Level Synthesis. One of them is the common statement of using fixed latency Functional Units.

However with the appearance of Variable Latency Functional Units, this statement has been broken, and a wide window of new possibilities has been opened. Functional Units can take a different number of cycles to complete an operation. This is the case of Speculative Functional Units, which are a concrete implementation of Variable Latency Functional Units whose latency depends on guessing some internal values. In this Ph.D Thesis two SFU designs have been proposed. Speculative Functional Units combine a good performance with a low area overhead, this is the reason why they can be so useful in High-Level Synthesis.

Nevertheless, Speculative Functional Units are useless if they cannot be effectively managed by synthesis tools. SFUs work faster when they guess the carry than when they suffer a misprediction, so the datapath controller must take this fact into account. Previous approaches, such as the *Telescopic*

Units [BMPM98], proposed to modify directly the controller by replicating states in those cases where these VLFUs were used. This will increase controllers' complexity exponentially and therefore their area penalty. Hence, only very few Telescopic Units can be used in every design.

In this Ph.D. Thesis two proposals have been done in order to tackle this problem. First, Centralized Management has been proposed in chapter 3. It consists in stopping the whole datapath everytime a misprediction happens, whatever the SFU. The main problem is that as the number of SFUs is increased, the probability of stalling the whole datapath will become higher. In order to let the datapath flow, the Distributed Management has been described in chapter 4. The idea is to stop only the operations that suffer the mispredictions, and those that depend on these, allowing the rest of them to continue their execution.

The area penalty due to these alternatives is small. And what is more, when considering multicycle FUs the Distributed Management keeps almost the same area as when utilizing monocycle FU. This fact compensates the area overhead with respect to the conventional implementations, that is also lower than the area penalty due to more complex implementations, such as the Carry Select Adders.

Experimental results confirm that with these two management techniques it is possible to integrate many SFUs while keeping a better performance than with the non-speculative implementations, specially with the Distributed Management technique.

In conclusion, a synthesis method has been proposed to allow the seamless deployment of Speculative Functional Units in High-Level Synthesis. Therefore, now that the *engineering* has been developed, the *architecture* can incorporate different *bricks*. In other words, now that the methodology has been established, new Speculative Functional Units can be automatically integrated into the High-Level Synthesis flow.

5.2. Contributions of this Ph. D. Thesis

Traditionally in High-Level Synthesis, the modules library has always been composed of fixed latency elements. Typically the best way to obtain performance has consisted in the use of complex Functional Units such as Carry Lookahead Adders, but increasing area and power overhead in exchange. However, in recent years the appearance of Variable Latency Functional Units allows us to optimize performance while not degrading the rest of the circuit parameters, thanks to the excellent tradeoffs that they possess in terms of area/power and performance.

Nevertheless, techniques presented in literature do not allow to fully take advantage of the Variable Latency Functional Units. This fact has motivated me to write this Ph.D. Thesis and contribute to the development of High-

Level Synthesis with the following issues:

(1) The design of Speculative Functional Units, which can be included in the modules library for their later use in the High-Level Synthesis flow.

Two Speculative Functional Units has been proposed in chapter 2. On the one hand, the use of a Predictive Adder has been introduced in order to mitigate the hardware overhead that presents other more complex structures such as the Carry Select Adders. When applied to linear adders, this technique allow to reduce 50 % of the overall delay, with negligible area overhead. When applied to logarithmic adders, it eliminates one level of carry propagation in exchange for a single predictor, which is only composed of one or few flipflops and some glue logic.

On the other hand, a Predictive Multiplier has also been proposed in chapter 2. The idea is to couple the Predictive Adder with a Carry Save like multiplier, which avoids the introduction of multiple predictors as in the Ripple Carry Multiplier. Finally, in order to deal with negative numbers, the Baugh-Wooley structure has been chosen.

(2) The introduction of prediction in the FUs design as an element able to reduce the critical path.

As it is observed in the Predictive or Speculative FUs presented in chapter 2, the delay is reduced by using structures similar to branch predictors, but in smaller scale. On the one hand, this helps to introduce Speculative Functional Units in the synchronous context, and besides it avoids delay elements that were used in previous asynchronous designs. On the other hand, prediction behaves well and it can be easily combined with forwarding structures if really high hit rates were necessary.

(3) The Speculative Execution Paradigm Theorem.

The analysis of the necessary conditions to commit an operation has been synthesized in this theorem. As the main idea of this Ph.D. Thesis is to let operations flow, the most important thing is to know when every operation can be written in its corresponding register, without violating the partial order imposed by the Dataflow Graph. Thus, an operation must guess the carry (condition 1), the local state controller must be the one associated with this operation (condition 2), and its

hazards must have been solved (condition 3). Condition 3 stems from the DFG itself. Condition 2 is inherent to the Distributed Management architecture. But condition 1 defines an interface to incorporate new Speculative Functional Units to the High-Level Synthesis flow. In other words, Speculative Functional Units can be implemented in many different ways, but they must always generate a hit signal for communicating with the controllers and being integrated in the architectures proposed by this Ph.D. Thesis.

(4) The definition of the Distributed Management architecture.

In order to handle several operations in different csteps, due to mispredictions, a local state controller per Speculative Functional Unit is necessary. There will be so many *state variables* as Speculative Functional Units. This is a considerable difference with the conventional datapath controllers, where there is only one state variable, because all the operations executed in a cycle will be synchronized with the same cstep, i.e. state.

Furthermore, these local state controllers must be synchronized. The Commit Signals Logic Unit is responsible of this, because it is the element that generates the enable signals that will fire the state transitions. The automatic generation of the Commit Signals Logic Unit for every Dataflow Graph is one of the main tasks of this Ph.D. Thesis.

(5) The incorporation of the Speculative Execution Paradigm to the Design Automation context.

The rules, structures and algorithms necessary for automatically generating every local state controller and the Commit Signals Logic Unit have been defined in chapter 4. A canonical architecture description for mapping the information given by the Dataflow Graph has been developed. In this way it is easy to translate this information into the target Hardware Description Language.

(6) The development of a simulation environment for the Centralized and Distributed Managements.

As Speculative Functional Units performance depends on actual values, a simulator was necessary. This is not the case of a conventional

execution, because in order to measure the execution time of an iteration it is enough to multiply the latency of the circuit, i.e. λ, by the cycle time. On the contrary, when using Speculative Functional Units, cycle time will be the same for every clock cycle, but the latency of every iteration will vary in function of mispredictions, i.e. events that happen in execution time.

(7) The development of new High-Level Synthesis techniques in order to improve the performance of speculative datapaths.

New Functional Unit and register binding policies have been presented in chapter 4 for increasing data correlation and thus hit rates, and for diminishing the number of hazards and then the innecessary stalls in the datapath.

5.3. Future lines of work

This Ph.D. Thesis is just a *building* without much *furniture*. The *pillars* have been established, but this initial construction creates some opportunities that must be taken advantage of.

The introduction of Speculative Functional Units in the High-Level Synthesis flow, obliges us to reconsider the traditional execution paradigm, which is non-speculative. Hence, the Speculative Execution Paradigm must be considered instead. This is the cradle for new techniques. For instance in chapter 4, new binding policies have been presented in order to increase performance, because they fit better to the special features of the Distributed Management.

Thereby, more ideas must be developed. They are currently in the author's mind, namely:

(1) The construction of a power/energy model in order to test the power/energy efficiency of the presented techniques.

(2) Multispeculative Functional Units. If with a predictor it is possible to reduce the execution time, will the use of several predictors help to reduce execution time much more?

(3) Dynamic binding. If dynamic scheduling is possible, why not dynamic binding? Can the movement of operations, from a FU to another one, reduce the average latency?

(4) Dynamic latency multicycle Functional Units. Will it make sense? Will it be possible to dynamically modify the latency of the Functional Units for some practical purpose, as reducing temperature, for instance?

In fact, some of them are actually on-going work. However they must be evaluated carefully and more time is required. Nonetheless, in this section a brief overview of these ideas will be given.

Note that appart from these future lines, another obvious ideas can be incorporated to the Distributed Management technique, such as the use of pipelined Functional Units or the inclusion of control dependencies in the Dataflow Graph.

5.3.1. Power and energy

In this Ph.D. Thesis the basis of the Speculative Execution Paradigm has been established. Afterwards, it has been shown that it is possible to operate with many Speculative Functional Units obtaining a good performance and with negligible area penalty. Observing the Distributed Management area results, one can intuitively suspect that power should not be increased very much. Therefore a power and energy detailed study is one of the most straightforward extensions of this work.

Two cases must be distinguished: with monocycle and with multicycle Functional Units. With monocycle FUs, the SFUs based implementations would increase frequency (only with respect to the RCA-like implementation), so power would increase linearly with respect to frequency. However, if execution time is lower than the conventional implementation, overall energy could be reduced. Nevertheless, as results with monocycle SFUs are not very encouraging, this option does not seem very promising.

With multicycle FUs, conventional and speculative cycle times are similar. Hence this power variation, due to frequency, should be negligible. As execution time is clearly lower than the RCA-like conventional case, overall energy should be lower too. When comparing with a CSEL-like conventional implementation, a finer analysis must be done. Carry Select Adders consumes more than 1.5 times than a Ripple Carry Adder, while the SFUs used in this Ph.D. Thesis should consume similar amount of power to a RCA if they hit, but when they fail only the most significant part is switching. Besides, the duplicated parts of the Carry Select Adders are always consuming leakage power. Hence, proposed SFUs should consume less power than CSELs. This fact joined with the smaller area penalty, points out that the Distributed Management implementation should consume less power than the CSEL one. Furthermore, as execution time is similar, and in some cases even smaller, this power reduction must be translated into an overall energy decrease.

5.3.2. Multispeculation

The second straightforward extension of this Ph.D. Thesis is the development of better Speculative Functional Units. The management techniques

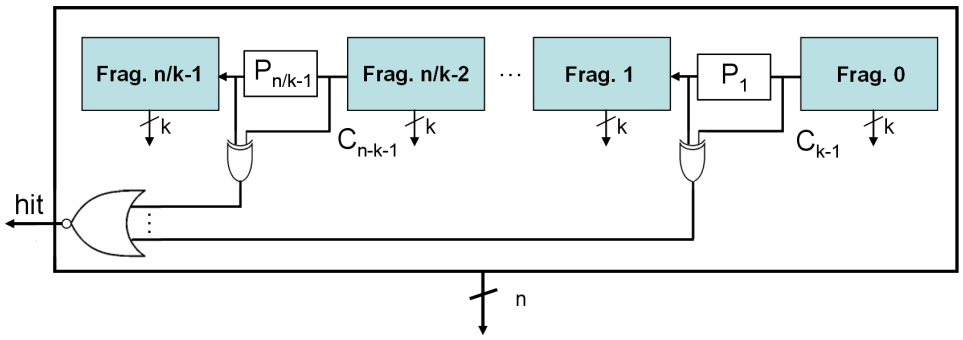


Figure 5.1: Multispeculative Adder general scheme

presented in this work allow the automatic integration of any SFU that complies with the interface requirements. In other words, a SFU receives two data inputs and produces a result and a hit signal, independently of how this SFU is implemented. The Distributed Controllers take these hit signals and then decide the next local states, regardless of the SFU implementations.

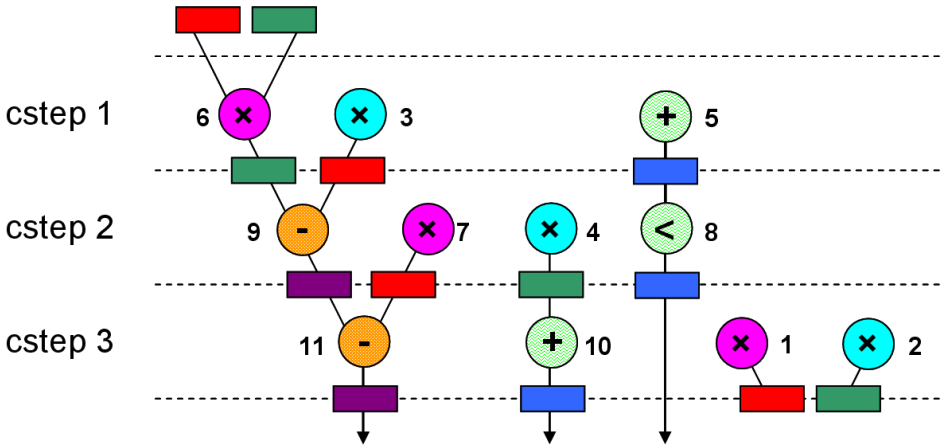
Hence, the next question is how to improve the Predictive Adders used in this Ph.D. Thesis (their improvement will produce an increase in multipliers performance too). Let's think about it. With a predictor, an n-bits Ripple Carry Adder reduces its delay from n to $n/2$. However this only happens if the predictor hits; if it fails another $n/2$ delay will be required for correcting the addition.

Therefore, what if the n-bits adder is implemented with several k-bits modules, $k \ll n$, that work in parallel ? If we suppose that predictors are located homogeneously as in figure 5.1, using k-bits fragments $n/k\text{-}1$ predictors would be necessary. Thereby, the adder delay would become proportional to the k-bits delay instead of the n-bits delay. But this will only happen if all the k predictors guess the corresponding carry. If one predictor fails, at least one more k-bits delay will be required. And what is more, a prediction failure could be propagated to a more significant fragment, and thus sucessively. Nevertheless according to the first performed experiments, this situation rarely happens if the size of the fragment is large enough, and almost all the additions are computed in two k-bits delays at the most.

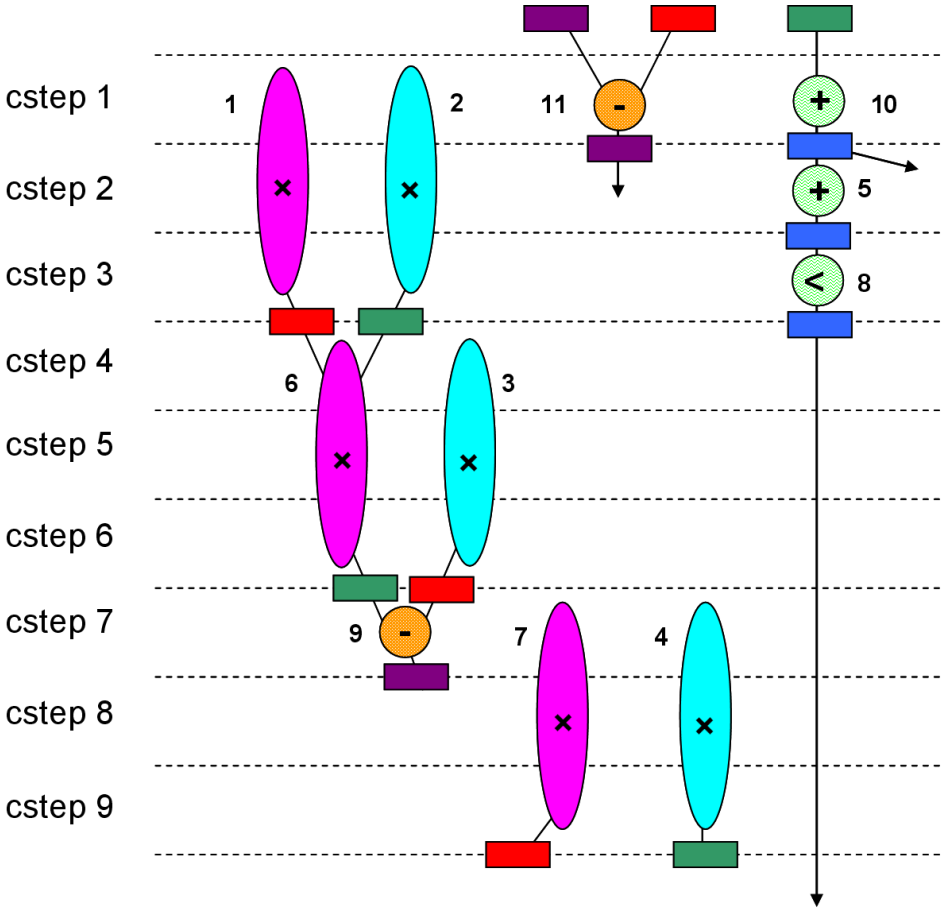
This is a very promising line that has been developed during the last months and that may produce good results in the short term.

5.3.2.1. Overcoming the limitation of Loop Folding

Loop folding or modulo scheduling [HHL91, KC97] is a scheduling technique that overlaps one or several iterations if possible, as software pipelining does in the compilers context [HP07]. It is composed of an initialization, body



(a) Monocycle FUs



(b) Multicycle FUs

Figure 5.2: DiffEq modulo scheduling using CSEL-like modules

and finalization periods. The body execution time will determine overall performance. See figure 5.2 for an example of loop folding.

Distributed Management implements a sort of dynamic loop folding when possible, i.e. when operations of the following iteration have solved their dependencies, they can be executed in parallel with those of the actual iteration. Hence, the Distributed Management execution flow will behave similar to loop folding when failures and dependencies allow it. So ... combining loop folding and Carry Select Adders will be better than Distributed Management?

In terms of performance loop folding combined with Carry Select Adders represent the upper bound of Distributed Management and proposed SFUs performance. Nevertheless, loop folding and CSELs have several counterparts. CSELs problem is area overhead, and thus power consumption will probably be a problem too, as it has been shown in the multicycle experiments in chapter 4. Loop folding obliges to modify the initial controller, and its utilization can produce a duplication of several states because of the initialization and finalization periods, and some additional registers could be necessary to store results proceeding from the overlapped iterations, then increasing area overhead. Besides, the latency gain of loop folding is not very significant if multicycle FUs are being considered. In the monocycle DiffEq implementation, shown by figure 5.2a, latency is reduced from 4 cycles to 3 cycles, i.e. 25 %, but in the multicycle, depicted by figure 5.2b only from 10 to 9, i.e. 10 %. These percentages are even worse if larger benchmarks are being considered.

Furthermore, the introduction of multiprediction will boost Distributed Management performance. Multispeculative Functional Units, as the SFUs proposed in this Ph.D. Thesis, cannot be handled with loop folding techniques. It is true that more CSEL levels can be included in every adder to compensate multiprediction, but it is also true that area penalty will increase. Besides, using *n/k-1* levels of CSEL in an n-bits adder does not guarantee a delay proportional to the k-bits modules, because of the additional delay caused by the intermediate multiplexers. And what is more, if the utilized basic blocks are logarithmic-like, e.g. Kogge-Stone adders, the area penalty will increase much more, because they must be duplicated.

5.3.3. Dynamic binding

Now that a sort of dynamic scheduling has been introduced in High-Level Synthesis as proposed in this Ph.D. Thesis, dynamic binding is another technique that should be tested, or at least thought over.

In section 4.1.2 several *Design Rules* were defined to certify the correctness of the circuits controlled with Dynamic Management. DR2.2 established that if the current operation was committed, the previous operation bound

to the same FU had been committed, while DR3.2 said that if the current operation could not be committed then the next operation bound to this FU could not be committed either.

However, both DR2.2 and DR3.2 are not indispensable. They has been created just to make it easier the controllers deployment. But there are some situations when an operation is waiting just because the controller has not reached the corresponding state. Dynamic binding would solve this situation. But a careful study is required in order to maintain a low area overhead and keep the correct state of the datapath.

5.3.4. Dynamic latency multicycle Functional Units

Distributed Management combined with multicycle Functional Units provides excellent results. On the one hand they achieve a good performance, while on the other hand area is very similar to the conventional implementations one.

With Distributed Management, the multicycle local controllers are based on the monocycle ones, so the Finite State Machines are quite simple. They only need the addition of a counter and some glue logic. This counter loads either the operating latency or the correction latency. However, conventional implementations need to add one state per cstep and generate the corresponding routing and load signals.

This means that the SFU local controllers are derived independently of the latency of the FUs, while conventional controllers depend on those latencies. Hence, the latency counter could be used for varying dynamically the latency of the modules. So, the only thing is to find a reason for doing it. For example, if a FU were getting hotter, the increase of its operating latency would help to diminish its temperature.

Finally the reader should note that all these dynamic techniques are only applicable and make sense if a dynamic scheduling is considered, at least from the performance point of view. For example, independently of the binding, a datapath will keep the same performance if it is using the Non Speculative Execution Paradigm, because the number of csteps is determined by the scheduling. Thus, dynamic binding makes no sense in a non-speculative context. Or for instance, modifying the latency of the FUs will only be possible in a dynamic scheduling context, because every FU latency change will change the initial static scheduling.

5.4. Publications

Here there is a list with the papers produced during my research:

(1) A.A. Del Barrio, S. Ogrenci Memik, M.C. Molina, J.M. Mendías and R. Hermida. "Power Optimization in Heterogeneous Datapaths". In Proc. *Design, Automation and Test in Europe (DATE)*. 2011, pp. 1400-1405.

(2) A.A. Del Barrio, S. Ogrenci Memik, M.C. Molina, J.M. Mendías and R. Hermida. "A Distributed Controller for Managing Speculative Functional Units in High Level Synthesis". In Proc. *IEEE Trans. On Computer-Aided Design of Integrated Circuits and Systems (TCAD)*. 2011, pp. 350-363.

(3) A.A. Del Barrio, S. Ogrenci Memik, M.C. Molina, J.M. Mendías and R. Hermida. "Using Speculative Functional Units in High Level Synthesis". In Proc. *Design, Automation and Test in Europe (DATE)*. 2010, pp. 1779-1784.

(4) M.C. Molina, R. Ruiz-Sautua, A.A. Del Barrio and J.M. Mendias. "Subword Switching Activity Minimization to Optimize Dynamic Power Consumption". *IEEE Design & Test of Computers*. 2009, pp. 68-77.

(5) A.A. Del Barrio, M.C. Molina, J.M. Mendías, E. Andrés, G. Botella, R. Hermida and F. Tirado. "Applying branch prediction techniques to implement adders". In Proc. *Conference on Design of Circuits and Integrated Systems (DCIS)*. 2008.

(6) A.A. Del Barrio, M.C. Molina, J.M. Mendías, E. Andrés, R. Hermida and F. Tirado. "Applying speculation techniques to implement functional units". In Proc. *IEEE International Conference on Computer Design (ICCD)*. 2008, pp. 74-80.

(7) A.A. Del Barrio, M.C. Molina, J.M. Mendías, E. Andrés and R. Hermida. "Restricted chaining and fragmentation techniques in power aware high level synthesis". In Proc. *Euromicro Conference on Digital System Design (DSD)*. 2008, pp. 267-273.

(8) E. Andrés, M.C. Molina, G. Botella, A.A. Del Barrio and J.M. Mendías. "Area Optimization of Combined Integer and Floating Point Circuits in High-Level Synthesis". In Proc. *Southern Programmable Logic Conference (SPL)*. 2008.

(9) E. Andrés, M.C. Molina, G. Botella, A.A. Del Barrio and J.M. Mendías. "Aerodynamics Analysis Acceleration Through Reconfigurable Hardware". In Proc. *Southern Programmable Logic Conference (SPL)*. 2008.

(10) A.A. Del Barrio, M.C. Molina, J.M. Mendías, and R. Hermida. "Pattern-guided switching minimization in high level synthesis". In Proc. *Conference on Design of Circuits and Integrated Systems (DCIS)*. 2007, pp. 175-180.

(11) A.A. Del Barrio, M.C. Molina and J.M. Mendías. "Bit-level Power Optimization during Behavioural Synthesis". In Proc. *Conference on Design of Circuits and Integrated Systems (DCIS)*. 2006.

(12) A.A. Del Barrio, M.C. Molina and J.M. Mendías. "Optimización a nivel de bit del consumo de potencia durante la Sintesis de Alto Nivel". In Proc. *Jornadas de Paralelismo*. 2006, pp. 621-626.

Appendix A

Performing Datapath simulations

Si se puede imaginar ... se puede programar :-)

Anónimo

Simulating values across the datapath is essential in order to check if there is a misprediction or not, because mispredictions will determine the stall of operations and therefore the final latency of the circuit.

Hence a simulator has been built in order to obtain the average latency of every benchmark. The overall execution time is the result of multiplying this average latency by the cycle time given by Synopsys Design Compiler. Note that the term *average latency* is being used instead of latency. As predictions depend on input data, different iterations of an algorithm may suffer different mispredictions, and thus may require a different number of cycles, i.e. a different circuit latency. Thereby, it is necessary to talk about average latency. In the simulations performed in this Ph.D. Thesis, 1,000 iterations of every benchmark have been run. More iterations were tried, but similar results were obtained.

In chapters 3 and 4 it is said that the simulator receives the specification of the algorithm and a characterization of the most common values of the primary inputs. This characterization is described in [BMMH07] and also utilized in [BMM+08b, MRSBM09]. The primary inputs characterization is generated synthetically. The corresponding characterization of the internal nodes can be either profiled or propagated thanks to a *pattern algebra* that will be described in this appendix. In the experiments presented in this Ph.D. Thesis the second option has been chosen, because it saves simulation time.

Nevertheless, data do not always behave as the assumed mean pattern and some random probability must be included to generate a more realistic

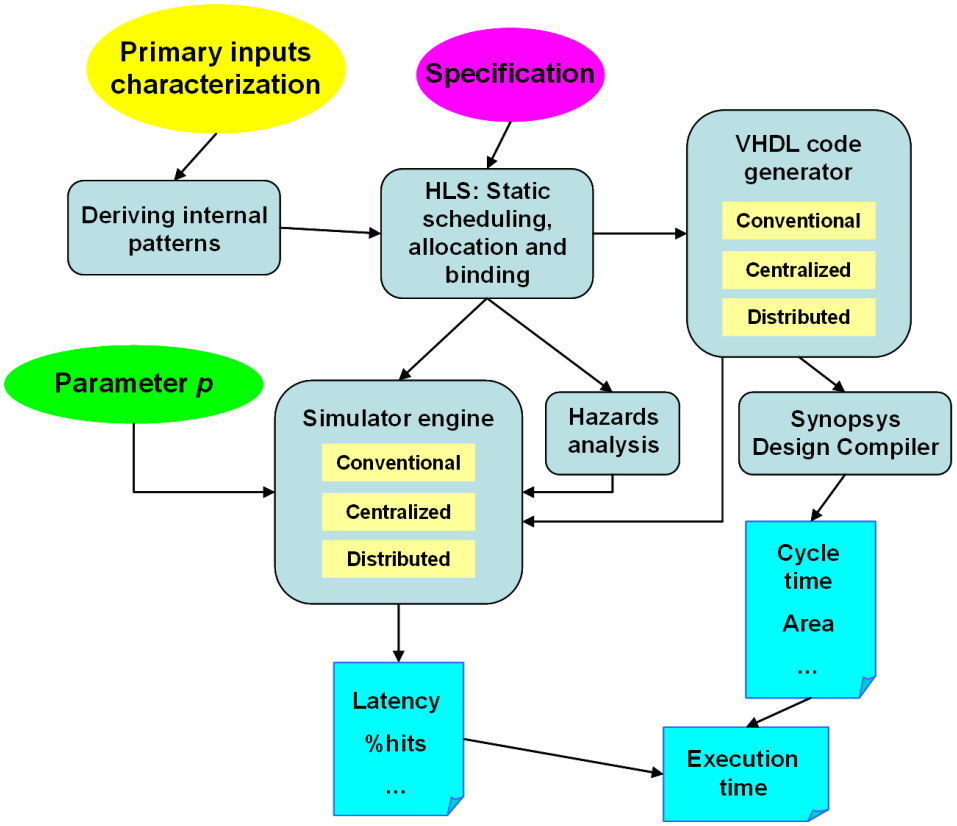


Figure A.1: Framework general scheme

simulation.

An overview of the whole system can be seen in figure A.1. The primary inputs characterization is utilized for deriving the internal nodes characterization. This information can be used for performing a scheduling, allocation or binding which takes it into account, e.g. the Hamming Distance bindings explained in section 4.5.2. Afterwards, the hazards analysis is performed in order to calculate the *Compatible States* of every operation. Then, the High-Level Synthesis information, the *Compatible States* and the p value are introduced to the engine of the simulator, which will run several iterations of the benchmark under consideration for every of the proposed architectures, and will produce different statistics such as the average latency, the hits percentage, etc.

On the other hand, the HLS information is also used for generating the VHDL code according to three different architectures: Conventional, Centralized (explained in chapter 3) and Distributed (explained in chapter 4). *Synopsys Design Compiler* will synthesize the circuit and will produce the area and timing reports. The cycle time given by *Synopsys* and the average

1^{st} operand	2^{nd} operand
0110	1111
0101	1110
0101	1100
1001	1011
0111	1010

Table A.1: Values profiled for two operands during a simulation

Pattern	1^{st} slot	2^{nd} slot
A	0	0
B	0	1
C	1	0
D	1	1

Table A.2: Patterns definition

latency given by the simulator will be multiplied to compose the execution time.

A.1. Patterns definition

As it has been said before, the patterns technique is first presented in [BMMH07] and subsequently used in [BMM+08b, MRSBM09]. However, this technique is an evolution of the *most common bits* technique presented in [BMM06a, BMM06b]. The most common bits are those which have the greatest probability to happen. In other words, if in the simulations the number of '1's is greater than the number of '0's, the the most common bit is '1'. Analogously for the opposite case.

The most common patterns technique is an extension of the most common bits, which divides the overall simulation time in several slots and applies the most common bits technique in every slot. Note that the number of slots is decided by the user, but if it is not too high, it is possible to use the alphabet letters to identify every pattern.

Supposing that the simulation time is divided in 2 slots, 4 patterns can be defined, as depicted by table A.2. For instance, pattern A means that in the first and second slots the most common bit is '0', pattern B means that in the first slot the most common bit is '0' and in the second is '1', and so on. If we suppose the profiled operands given by table A.1, it is possible to compute the most common patterns of every bit in every operand following the procedure depicted in table A.3. For every operand, the columns labeled

Bit	1^{st} slot	Prob	2^{nd} slot	Prob	MCP
3	0	3/3	1	1/2	B
2	1	3/3	1	1/2	D
1	0	2/3	1	1/2	B
0	1	2/3	1	2/2	D

(a) Patterns calculation 1^{st} operand

Bit	1^{st} slot	Prob	2^{nd} slot	Prob	MCP
3	1	3/3	1	2/2	D
2	1	3/3	0	2/2	C
1	1	2/3	1	2/2	D
0	0	2/3	1	1/2	B

(b) Patterns calculation 2^{nd} operand

Table A.3: Patterns calculation

as *1st* and *2nd slots* are the most common bits in the corresponding slot. This can be checked with the columns labeled as *Prob*, where the probability of obtaining the corresponding most common bit is anotated.

For example, in the third bit of the second operand the most common bit is '1' in both slots, so the final pattern for this bit will be the pattern D. In the first slot, '1' occurs in 3 of 3 possible cases, while in the second slot it occurs in 2 of 2 possible cases. Note that when 50 % of probability is reached, a '1' most common bit has been assumed. This fact will rarely happen, because it is very difficult to have an exact 50 % probability. Nevertheless, in some cases of the example this probability happens because of the reduced set of profiled values.

A.2. Defining a patterns algebra

Defining a pattern algebra is really important to propagate patterns to the internal nodes of the Dataflow Graphs. In this way, it is not necessary to simulate the whole design if the primary inputs patterns are known.

As patterns are based on the combination of most common bits, it is possible to create a Boolean-like algebra just by defining the operations of a universal gates set [HCPS98]. There are several universal gates sets, but the one composed by the basic logic operations **{AND, OR, NOT}** has been chosen.

Let's suppose two patterns composed of a set of most common bits, namely: $P=p_{n-1}...p_1p_0$ and $Q=q_{n-1}...q_1q_0$. Besides let's suppose a function *ToPattern*, defined by equation A.1, which transforms a bits chain into a

pattern.

$$\begin{aligned} &ToPattern : \{Bits\} \rightarrow \{Patterns\} \\ &ToPattern(p_{n-1} \ \ldots \ p_1 p_0) = P \end{aligned} \tag{A.1}$$

Note that this function is a bijective application, because for every bits chain there only exists a pattern and for every pattern there only exists a corresponding bits chain.

The equations that define the universal gates set for patterns are shown below. Note that the sign & means the operator concatenation.

$$P \wedge Q = ToPattern(p_{n-1} \wedge q_{n-1} \ \& \ \ldots \ \& \ p_1 \wedge q_1 \ \& \ p_0 \wedge q_0) \tag{A.2}$$

$$P \vee Q = ToPattern(p_{n-1} \vee q_{n-1} \ \& \ \ldots \ \& \ p_1 \vee q_1 \ \& \ p_0 \vee q_0) \tag{A.3}$$

$$\neg P = ToPattern(\neg p_{n-1} \ \& \ \ldots \ \& \ \neg p_1 \ \& \ \neg p_0) \tag{A.4}$$

As it can be observed in equations A.2, A.3 and A.4, the logic operation under question is applied to every most common bit and then the results are concatenated to form a bits chain. Finally this bits chain is transformed into the corresponding pattern.

As with any universal gates set it is possible to specify any combinational function, the equations of the adders, subtracters, multipliers, etc. can be expressed as a combination of these basic logic operations. For example, see all the logic equations utilized in chapter 2 for modelling the behavior of every FU. The only difference will be the inputs to these logic equations: instead of using bits, the patterns will be utilized.

For example, let's see in depth the addition of the patterns AABD+AACC. First, let's suppose the same patterns as in table A.2. Second, the reader must remember the equations of the Ripple Carry Adder, shown in equation A.5 and explained in section 2.1.1.

$$\begin{aligned} s_i &= x_i \oplus y_i \oplus c_i \\ c_{i+1} &= x_i \cdot y_i + c_i \cdot (x_i + y_i) \end{aligned} \tag{A.5}$$

Note that the carry-in to the addition is '0', so in terms of patterns this carry-in will be the pattern A (all '0's). The rest of the carry-in's are the

carry-out's from the previous stage. The development of this calculus can be followed in equation A.6

$$\begin{aligned}
s_0 &= D \oplus C \oplus A = 11 \oplus 10 \oplus 00 = 01 \oplus 00 = 01 = B \\
c_1 &= D \cdot C + A \cdot (D + C) = 11 \cdot 10 + 00 \cdot (11 + 10) = 10 = C \\
s_1 &= B \oplus C \oplus C = 01 \oplus 10 \oplus 10 = 11 \oplus 10 = 01 = B \\
c_2 &= B \cdot C + C \cdot (B + C) = 01 \cdot 10 + 10 \cdot (01 + 10) = 10 = C \\
s_2 &= A \oplus A \oplus C = 00 \oplus 00 \oplus 10 = 00 \oplus 10 = 10 = C \\
c_3 &= A \cdot A + C \cdot (A + A) = 00 \cdot 00 + 10 \cdot (00 + 00) = 00 = A \\
s_3 &= A \oplus A \oplus A = 00 \oplus 00 \oplus 00 = 00 \oplus 00 = 00 = A \\
c_4 &= c_{out} = A \cdot A + A \cdot (A + A) = 00 \cdot 00 + 00 \cdot (00 + 00) = 00 = A
\end{aligned} \tag{A.6}$$

As it can be observed AABD+AACC = ACBB, with an A pattern as carry-out.

The rest of the operations can be computed in the same way. Therefore the internal nodes patterns can be generated and thus utilized for High-Level Synthesis purposes, such as the Hamming Distance bindings explained in section 4.5.2.

A.3. The importance of parameter *p*

The bit level characterization of the primary inputs is one of the simulator inputs. Nevertheless, as real input values are not usually available for each benchmark, these patterns have been generated synthetically. Afterwards the internal nodes patterns have been calculated with the aforementioned algebra, and can be used to apply any static High-Level Synthesis technique. However, the average values captured by the patterns are not always repeated. Hence, in order to perform a more realistic study, it is necessary to include some probability p while introducing the primary inputs to the Dataflow Graph every iteration. Therefore, p will measure the probability of the pattern to behave as it was assumed.

In order to illustrate how these primary input values are generated, the example depicted in table A.4 has been developed. Let's suppose that there is a pattern AAABDDCC for a certain operand. And let's suppose the patterns definition given by table A.2, and p=0.75. Overall simulation time will be 4 iterations, so the first slot will range from iteration 1 to iteration 2, and the second slot from iteration 3 to iteration 4.

Table A.4 is composed of three columns. The first one indicates the iteration. The second contains an array of eight random values between 0 and 1, i.e. so many *coin* values as the operand length. The third column is the final bits chain that is generated. This third column combines table A.2 with the

Input pattern = AAABDDCC		
Iteration	Coins vector	Bits chain
1	[0.2, 0.3, 0.4, 0.7, 0.9, 0.1, 0.8, 0.6]	00000101
2	[0.7, 0.5, 0.9, 0.8, 0.6, 0.4, 0.1, 0.2]	00111111
3	[0.3, 0.7, 0.5, 0.1, 0.3, 0.3, 0.7, 0.9]	00011101
4	[0.1, 0.9, 0.1, 0.2, 0.4, 0.6, 0.3, 0.7]	01011100

Table A.4: Primary input bits generation for pattern AAABDDCC

iterations and the *coin* values. The procedure consists in using the pattern value corresponding to a concrete slot, and if $coin \leq p$ the generated bit will be this value. Otherwise, i.e. $coin > p$, the generated bit will be the opposite value to that.

For example, in iteration 1 the A pattern of the 7^{th} position will be translated into '0', because $0.2 \leq 0.75$, and in the first slot the most common bit was '0'. In the same iteration, the pattern D of the 4^{th} will be translated into '0', because $0.9 > 0.75$ and the assumed value was '1', according to table A.2. In iteration 3, i.e. second slot, the B pattern will be translated into '1', because $0.1 \leq 0.75$ and B="01", according to the mapping table. The C pattern of position 0 will be translated into '1' because $0.9 > 0.75$ and C="10". And so on.

This is the procedure for generating the primary input bits. Finally, these bits will be used for simulating the rest of the values along the entire datapath, i.e. through the internal nodes. This is done in a similar fashion to the pattern calculation, but with single bits, i.e. with a common Boolean algebra. In this way, datapaths can be simulated at Register Transfer Level level and with real values.

Appendix B

Compatible States Calculation

An algorithm must be seen to be believed

Donald Knuth

In chapter 4 the *Compatible States* are defined in the Distributed Management context as a method to check if the hazards between two operations, one in execution, i.e. O_e, and one which causes the dependency, i.e. O_d, have been solved.

It is necessary to check if O_d has been committed, or at least if it can be committed in the WAR hazards case, in order to commit O_e. The method proposed in this Ph.D. Thesis consists in evaluating if the local state associated to the Speculative Functional Unit where O_d has been bound, i.e. St_d, is later than the symbolic state value associated to O_d, i.e. S_d. In this way, if St_d is later than the state value which corresponds with O_d, the hazard will have been solved, because this will imply that O_d has been committed. Note that the case of WAR hazards is special, as explained in chapter 4, but it has no effect over the calculation of the *Compatible States*, which is the purpose of this appendix.

Hence, in order to check dependencies it is necessary to evaluate the local states. However, there can be many states values to compare, specially if the circuit is large. This can lead to an unnecessary area overhead, because not every state value can be taken by St_d, if O_e is being executed. That is the reason why *Compatible States* were defined in section 4.1.2.1 as those states that *can happen*. This fact can be used to simplify the number of possible state values, and thus reduce the area penalty due to the checking conditions.

In this appendix, a detailed example of how *Compatible States* are calculated will be fully developed in order to make their understanding easier.

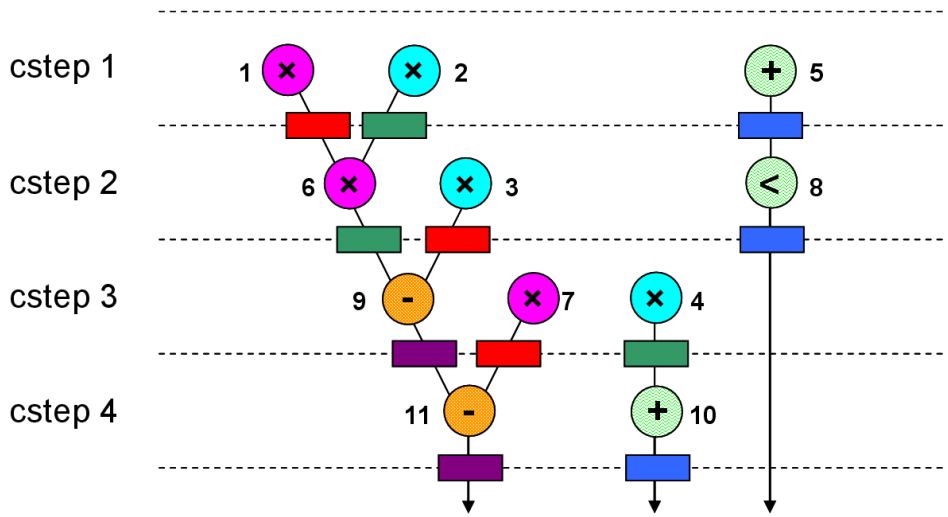


Figure B.1: DiffEq scheduling and FU and register binding

cstep	M1	M2	A1	A2	R1	R2	R3	R4
1	1	2	5		1	2	5	
2	6	3	8		3	6	8	
3	7	4		9	7	4		9
4			10	11			10	11

Table B.1: DiffEq binding summary

B.1. Example of *Compatible States* Calculation

Before going deeper into the example, the reader must remember the pseudocode that corresponds to the *Compatible States* calculation, which is shown in algorithm B.1, and the DiffEq scheduling and binding, depicted by figure B.1 and table B.1.

Basically, as it can be observed in algorithm B.1 the idea consists in discarding those states that cannot happen, because based on the DFG analysis it is completely sure that those states will not happen. For instance, if *Operation 7* is being executed, it is mandatory that *Operation 6* has been committed before, because according to the Design Rule DR2.2 it must have been committed.

The calculation of the *Compatible States* of M2 while executing *Operation 6* will be developed as an example of the application of the algorithm. *Operation 6*, executed in M1, has a RAW hazard with *Operation 2*, executed in M2. The question that must be solved is if *Operation 2* has been committed, i.e. St_{M2} ">" S2. Therefore, the *Compatible States* of M2, while executing

Algorithm B.1 getCompatibleStates(St_e,St_d)

1: $Comm \leftarrow getCommittedStates(St_e)$
2: $NComm \leftarrow getNonCommittableStates(St_e)$
3: $CS1 \leftarrow \{St : St \in \{St_{FUd}\} \wedge St \neq St_d \wedge St \notin Comm\}$
4: $CS2 \leftarrow \{St : St \in \{St_{FUd}\} \wedge St \neq St_d \wedge St \notin NComm\}$
5: $CS \leftarrow CS1 \bigcup CS2$
6: $CS \leftarrow CS \bigcup \{First\ St \in \{St_{FUd}\} : St \neq St_d\ \wedge St \in NComm\}$
7: **return** CS

Operation 6 in M1, must be calculated.

(1) *Comm* ← *Committed States* in an iteration backwards

This step is performed according to the Design Rules DR1 and DR2, presented in section 4.1.2. We look for those states that have been committed in an iteration backwards. Note that a window size of an iteration is enough, because if an operation has been committed in iteration *(i-1)*, it has been committed in all the previous iterations. If C_e is the cstep where O_e was statically scheduled, and i is the actual iteration, then the concrete window interval would be [$C_e{}^{i\text{-}1}$, $C_e\text{-}1^i$]. The pseudocode of this method is shown in algorithm B.2.

Then, as *Operation 6* is being executed, it is possible to infere that *Operation 1* has been committed because it is the previous operation bound to M1, according to DR1. As *Operation 1* has been committed, *Operation 11* has been committed too, because of the WAR hazard, according to DR2.4. Note that *Operation 11* is located in the previous iteration. As *Operation 11* has been committed, *Operations 9* and *7* have been committed too, because of the RAW hazards, according to DR2.1. Note that it is also possible to deduce that *Operation 7* has been committed because it is bound to the same FU than *Operation*

Algorithm B.2 getCommittedStates(St_e)

1: $Comm \leftarrow getStatesThatSatisfy(DR1, St_e)$
2: $ProcessedStates \leftarrow Comm$
3: **while** ProcessedStates has more elements **do**
4: $St_h \leftarrow Head(ProcessedStates)$
5: **if** $St_h \notin Comm$ **then**
6: $Comm \leftarrow Comm \bigcup getStatesThatSatisfy(DR2, St_h)$
7: **end if**
8: **end while**
9: **return** $Comm$

1, satisfying DR2.2. As *Operation 9* has been committed, *Operation 4* has also been committed because of the WAR hazard, according to DR2.4.

Hence *Comm* = { $S1^{i}$, $S11^{i-1}$, $S9^{i-1}$, $S7^{i-1}$, $S4^{i-1}$ }

Note that the superscript indicates the iteration corresponding with every state. This index satisfies the *Iterations Theorem*, explained in section 4.2.3.

(2) *NComm* ← *Non-Committable States* in an iteration forwards

The calculation of the non-committable states is performed one iteration forwards. Applying the same reasoning as in the committed states, if an operation cannot be committed in iteration *(i+1)*, it will not be committed in all the following iterations. The Design Rules DR3 and DR4 are utilized for deriving those states that cannot be committed. In this way the non satisfaction of the RAW, WAR and WAW hazards with later operations can be derived. The considered window would be defined by the interval [$C_e{}^{i}$, C_e-1^{i+1}]. The pseudocode of this method is depicted by algorithm B.3

According to DR3.4, if *Operation 6* is just being executed, *Operation 3* cannot be committed either, because of a WAR hazard. As *Operations 6* and *7* are bound to M1 , *Operation 7* will not be committed, according to DR3.2. If *Operation 6* is just being executed, *Operation 9* cannot be committed because of the RAW hazard, according to DR3.1. If *Operation 9* is non-committable, *Operation 4* will not be committed either, because of the WAR hazard and according to DR4.4. If *Operations 9* and *4* are non-committable, then *Operations 11* and *10* will not be committed either, respectively, because of the RAW hazards, according to DR4.1. According to DR4.2, due to structural hazards, neither *Operations 1*, *2*, or *5* will be committed. Note that they belong to iteration *(i+1)*.

Algorithm B.3 getNonCommittableStates(St_e)

```
1: NComm ← getStatesThatSatisfy(DR3, St_e)
2: ProcessedStates ← Comm
3: while ProcessedStates has more elements do
4:    St_h ← Head(ProcessedStates)
5:    if St_h ∉ NComm then
6:       NComm ← NComm ⋃ getStatesThatSatisfy(DR4, St_h)
7:    end if
8: end while
9: return NComm
```

Hence $NComm$ = { $S3^i$, $S9^i$, $S7^i$, $S4^i$, $S11^i$, $S10^i$, $S1^{i+1}$, $S2^{i+1}$, $S5^{i+1}$ }

(3) $CS1 \leftarrow$ States of FU_d controller $\neq St_d$ that do not belong to the *Committed States*

The only M2 state different to S2 that does not belong to this list is S3, because S4 does.

$CS1$ = { $S3^i$ }

(4) $CS2 \leftarrow$ States of FU_d controller $\neq St_d$ that do not belong to the *Non-Committable States*

There is no M2 state in this list, because both S3 and S4 belong to the non-committable states list.

$CS2$ = { }

(5) $CS \leftarrow CS1 \bigcup CS2$

CS = { $S3^i$ }

(6) $CS \leftarrow CS \bigcup$ (First State of FU_d controller $\neq St_d$ that belongs to the *Non-Committable States*)

S3 is the first M2 state inside the non-committable states list. Thereby, it can be executed, i.e. reached, but not committed.

CS = { $S3^i$ }

(7) return CS

Therefore the *Compatible States* of M2, while executing *Operation 6* in M1, will be composed of $S3^i$. In other words, the RAW hazard between *Operations 6* and *2* will be solved just by checking if St_{M2}=$S3^i$. The iteration property i means that S3 must be in the same iteration than *Operation 6*. Thereby, according to the Distributed Management iterations control explained in subsection 4.2.3, the actual condition would be St_{M2}=S3 $\wedge$ $iteration_{M1}$=$iteration_{M2}$.

As it can be observed, the complexity of the algorithm is dominated by steps 1 and 2. If n is the number of operations, in these steps all the operations in a window of an iteration are considered. Hence, the complexity of steps 1 and 2 is $\mathbf{O(n)}$. On the other hand, this *Compatible States* calculation will be performed for every hazard and for every operation, so if the

mean number of hazards is $\bar{H}$, the number of executions of the algorithm will be in $\mathbf{O}(\overline{H}\,\boldsymbol{n})$. However, as the number of hazards will usually be much lower than n, the number of the executions will be comprised between $\mathbf{O}(\boldsymbol{n})$ $< \mathbf{O}(\overline{H}\,\boldsymbol{n}) \ll \mathbf{O}(\boldsymbol{n^2})$, but closer to $\mathbf{O}(\boldsymbol{n})$. Therefore, the overall complexity of the *Compatible States* calculation of the whole Dataflow Graph will be $\mathbf{O}(\overline{H}\,\boldsymbol{n^2})$, but close to $\mathbf{O}(\boldsymbol{n^2})$.

Bibliografía

En resolución, él se enfrascó tanto en su lectura, que se le pasaban las noches leyendo de claro en claro, y los días de turbio en turbio, y así, del poco dormir y del mucho leer, se le secó el cerebro, de manera que vino a perder el juicio. Llenósele la fantasía de todo aquello que leía en los libros, así de encantamientos, como de pendencias, batallas, desafíos, heridas, requiebros, amores, tormentas y disparates imposibles, y asentósele de tal modo en la imaginación que era verdad toda aquella máquina de aquellas soñadas invenciones que leía, que para él no había otra historia más cierta en el mundo

El Ingenioso Hidalgo don Quijote de la Mancha, Miguel de Cervantes Saavedra

[ADH05] E.M. Ashmila, S. Dlay, and O. Hinton. Adder methodology and design using probabilistic multiple carry estimates. *IEE Proc. Computers and Digital Techniques*, 152:697–703, 2005.

[ARI10] ARITH research group, Aoki laboratory, Tohoku University. Hardware algorithms for arithmetic modules. `http://www.aoki.ecei.tohoku.ac.jp/arith/mg/algorithm.html`, 2010.

[AST67] D.W. Anderson, F.J. Sparacio, and R.M. Tomasulo. Ibm 360/91: Machine philosophy and instruction-handling. *IBM Journal of Research and Development*, 11:8–24, 1967.

[Bar81] M. Barbacci. Instruction set processor specifications (isps): The notation and its applications. *IEEE Transactions on Computers*, 30:24–40, 1981.

[BCK09] D. Bañeres, J. Cortadella, and M. Kishinevsky. Variable-latency design by function speculation. In *Proceedings Of Design, Automation and Test in Europe*, pages 1704–1709, 2009.

[Bed62] O.J. Bedrij. Carry-select adder. *IRE Transactions on Electronics Computers*, 11:340–346, 1962.

[Ber95] Berkeley Design Technology, Inc. *Proceedings Of the 1995 DSPx Exhibition and Symposyum*, 1995.

[BK82] R.P. Brent and H.T. Kung. A regular layout for parallel adders. *IEEE Transactions on Computers*, 31:260–264, 1982.

[BKI99] B. Bishop, T.P. Kelliber, and M.J. Irwin. A detailed analysis of mediabench. In *Workshop on Signal Processing Systems*, pages 448–455, 1999.

[BMM06a] A.A.D. Barrio, M.C. Molina, and J.M. Mendías. Bit-level power optimization during behavioral synthesis. In *Proceedings Of the 21th Conference on Design of Circuits and Integrated Systems*, 2006.

[BMM06b] A.A.D. Barrio, M.C. Molina, and J.M. Mendías. Optimización a nivel de bit del consumo de potencia durante la síntesis de alto nivel. In *Proceedings Of the 17th Jornadas de Paralelismo*, pages 621–626, 2006.

[BMM+08a] A.A.D. Barrio, M.C. Molina, J.M. Mendías, E. Andrés, G. Botella, R. Hermida, and F. Tirado. Applying branch prediction techniques to implement adders. In *Proceedings Of the 23th Conference on Design of Circuits and Integrated Systems*, 2008.

[BMM+08b] A.A.D. Barrio, M.C. Molina, J.M. Mendías, E. Andrés, and R. Hermida. Restricted chaining and fragmentation techniques in power aware high level synthesis. In *Proceedings Of 11th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*, pages 267–273, 2008.

[BMM+08c] A.A.D. Barrio, M.C. Molina, J.M. Mendías, E. Andrés, R. Hermida, and F. Tirado. Applying speculation techniques to implement functional units. In *Proceedings Of IEEE International Conference on Computer Design*, pages 74–80, 2008.

[BMM+10] A.A.D. Barrio, M.C. Molina, J.M. Mendías, R. Hermida, and S. Ogrenci Memik. Using speculative functional units in high level synthesis. In *Proceedings Of Design, Automation and Test in Europe*, pages 1779–1784, 2010.

[BMM+11] A.A.D. Barrio, S. Ogrenci Memik, M.C. Molina, J.M. Mendías, and R. Hermida. A distributed controller for managing speculative functional units in high level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30:350–363, 2011.

[BMMH07] A.A.D. Barrio, M.C. Molina, J.M. Mendías, and R. Hermida. Pattern-guided switching minimization in high level synthesis. In *Proceedings Of the 22th Conference on Design of Circuits and Integrated Systems*, pages 175–180, 2007.

[BMPM98] L. Benini, E. Macii, M. Poncino, and G. De Micheli. Telescopic units: A new paradigm for performance optimization of vlsi design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17:220–232, 1998.

[Boo51] A.D. Booth. A signed binary multiplication technique. *Quarterly Journal of Mechanics and Applied Mathematics*, 4:236–240, 1951.

[BS73] M. Barbacci and D. Siewiorek. Automated exploration of the design space for register transfer (rt) systems. In *Proceedings of the 1st Annual Symposium on Computer Architecture*, pages 101–106, 1973.

[BS74] M. Barbacci and D. Siewiorek. *Some Aspects of the Symbolic Manipulation of Computer Descriptions*. Department of Computer Science, Carnegie-Mellon University, 1 edition, 1974.

[BSG+77] M. Barbacci, D. Siewiorek, R. Gordon, R. Howbrigg, and S. Zuckerman. An architectural research facility: Isp descriptions, siulation, data collection. In *Proceedings of the 1977 National Computer Conference*, pages 161–173, 1977.

[BW73] C.R. Baugh and B.A. Wooley. A two's complement parallel array multiplication algorithm. *IEEE Transactions on Computers*, 22:1045–1047, 1973.

[Cha99] M. Charrier. Jpeg2000, the next millennium compression standard for still images. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, pages 131–132, 1999.

[Cil09] A. Cilardo. A new speculative addition architecture suitable for two's complement operations. In *Proceedings Of Design, Automation and Test in Europe*, pages 664–669, 2009.

[CM08] Phillip Coussy and Adam Morawiec. *High-Level Synthesis. From Algorithm to Digital Circuit.* Springer Netherlands, 1 edition, 2008.

[CR89] R. Camposano and W. Rosenstiel. Synthesizing circuits from behavioral descriptions. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8:171–180, 1989.

[CS02] C. Chen and M. Sarrafzadeh. Power-manageable scheduling technique for control dominated high-level synthesis. In *Proceedings of the Design, Automation and Test in Europe*, pages 1016–1020, 2002.

[CT89] R. Camposano and R. Tabet. Design representation for the synthesis of behavioral vhdl models. In *Proceedings of the 9th International Symposium on Computer Hardware Description Languages and tehr Applications*, pages 49–58, 1989.

[CW91] R. Camposano and W. Wolf. *High Level VLSI Synthesis.* Kluwerl, 1991.

[Dad65] L. Dadda. Some schemes for parallel multipliers. *Alta Frequenza*, 34:346–356, 1965.

[Dad76] L. Dadda. On parallel digital multipliers. *Alta Frequenza*, 45:574–580, 1976.

[DeJ91] G. DeJong. Data flow graphs: System specification with the most unrestricted semantics. In *Proceedings of the European Design Automation Conference*, pages 401–405, 1991.

[DWA+92] D.W. Dobberpuhl, R.T. Witek, R. Allmon, R. Anglin, D. Bertucci, S. Britton, L. Chao, R.A. Conrad, D.E. Dever, B. Gieseke, S.M.N. Hassoun, G.W. Hoeppner, K. Kuchler, M. Ladd, B.M. Leary, L. Madden, E.J. McLellan, D.R. Meyer, J. Montanaro, D.A. Priore, V. Rajagopalan, S. Samudrala, and S. Santhanam. A 200 MHz 64-b dual-issue cmos microprocessor. *IEEE Journal of Solid-State Circuits*, 27:1555–1565, 1992.

[GE92] C.H. Gebotys and M.I. Elmasry. Optimal synthesis of high-performance architectures. *IEEE Journal of Solid State Circuits*, 27:389–397, 1992.

[Gir84] E. Girczyc. *Automatic Generation of Micro-sequenced Data Paths to Realize ADA Circuit Descriptions.* Phd, Carleton University, 1984.

[HC87] T. Han and D.A. Carlson. Fast area-efficient vlsi adders. In *Proceedings of 8th Symposium on Computer Arithmetic*, pages 49–56, 1987.

[HCPS98] R. Hermida, A.M. Del Corral, E. Pastor, and F. Sánchez. *Fundamentos de Computadores.* Editorial Síntesis, 1 edition, 1998.

[HHL91] C-T. Hwang, Y-C. Hsu, and Y-L. Liu. Scheduling for functional pipelining and loop winding. In *Proceedings of the 28th ACM/IEEE Design Automation Conference*, pages 764–769, 1991.

[HJH+93] S. Huang, Y.L. Jeang, C.T. Hwang, Y.C. Hsu, and J.F. Wang. A tree-based scheduling algorithm for control-dominated circuits. In *Proceedings of the 30th international Design Automation Conference*, pages 578–582, 1993.

[HLH91] C.T. Hwang, T.H. Lee, and Y.C. Hsu. A formal approach to the scheduling problem in high level synthesis. *IEEE Transactions on Computer-Aided Design*, 10:464–475, 1991.

[HP07] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 4 edition, 2007.

[JAC02] J. Jeon, Y. Ahn, and K. Choi. Cdfg toolkit user's guide. Technical report, School of Electronic Engineering, Seoul National University, 2002.

[Jan01] Jack Jansen. Adaptive differential pulse code modulation. `http://web.archive.org/web/20021005133648/http://www.cs.ucla.edu/~leec/mediabench/applications.html#ADPCM`, 2001.

[KC97] D. Kim and K. Choi. Power-conscious high level synthesis using loop folding. In *Proceedings of the 34th Design Automation Conference*, pages 441–445, 1997.

[KMO+08] T. Kato, T. Miyauchi, Y. Osumi, H. Yamauchi, H. Nishikado, T. Miyake, and S. Kobayashi. A cdfg generating method from c program for lsi design. In *Proceedings of the IEEE Asia Pacific Conference on Circuits and Systems*, pages 936–939, 2008.

[Kor02] I. Koren. *Computer Arithmetic Algorithms.* A K Peters, 2 edition, 2002.

[KR07] I. Kuon and J. Rose. Measuring the gap between asic and fpgas. *IEEE Transactions on Computer-Aided Design*, 26:203–215, 2007.

[KS73] P.M. Kogge and H.S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Transactions on Computers*, 22:786–793, 1973.

[KT85] T.J. Kowalski and D.E. Thomas. The VLSI design automation assistant: what's in a knowledge base. In *Proceedings of the 22nd ACM/IEEE Design Automation Conference*, pages 252–258, 1985.

[Lab03] Microelectronic Embedded Systems Laboratory. Spark parallelizing high-level synthesis framework. `http://mesl.ucsd.edu/spark`, 2003.

[LF80] R.E. Ladner and M.J. Fischer. Parallel prefix computation. *Journal of ACM*, 27:831–838, 1980.

[Lin81] H. Ling. High speed binary adder. *IBM Journal of Research and Developement*, 25:156–166, 1981.

[LJ92] T. Lynch and E.E. Swartzlander Jr. A spanning tree carry look-ahead adder. *IEEE Transactions on Computers*, 41:931–939, 1992.

[LL00a] K.M. Lepak and M.H. Lipasti. On the value locality of store instructions. In *Proceedings of 27th Annual International Symposium on Computer Architecture*, pages 182–191, 2000.

[LL00b] T. Liu and S. Lu. Performance improvement with circuit level speculation. In *Proceedings of 33th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 348–355, 2000.

[LM09] J. Luiz and V. Martins. C commands implemented direct into the hardware using the chipcflow machine. In *IX Jornadas de Computación Reconfigurable y Aplicaciones*, pages 215–224, 2009.

[LMD94] B. Landwehr, P. Marwedel, and R. Doemer. Oscar: Optimum simultaneous scheduling, allocation and resource binding based on integer programming. In *Proceedings of the conference on European design automation*, pages 90–95, 1994.

[LWS96] M.H. Lipasti, C.B. Wilkerson, and J.P. Shen. Value locality and load value prediction. In *Proceedings of 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 138–147, 1996.

[Mar86] P. Marwedel. A new synthesis for the mimola software system. In *Proceedings of the 23rd ACM/IEEE Design Automation Conference*, pages 271–277, 1986.

[Mic94] G. De Micheli. *Synthesis and Optimization of Digital Circuits.* McGraw-Hill, 1 edition, 1994.

[MLS96] T. Mudge, C.C. Lee, and S. Sechrest. Correlation and aliasing in dynamic branch predictors. In *Proceedings of the 23rd International Symposium on Computer Architecture*, pages 22–32, 1996.

[MM05] S. Mondal and S. Ogrenci Memik. Resource sharing in pipelined cdfg synthesis. In *Proceedings of the Asia and South Pacific Design Automation Conference*, pages 795–798, 2005.

[Moo65] G.E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38:114–117, 1965.

[MPC88] M.C. McFarland, A.C. Parker, and R. Camposano. Tutorial on high-level synthesis. In *Proceedings of the 25th ACM/IEEE Design Automation Conference*, pages 330–336, 1988.

[MRSBM09] M.C. Molina, R. Ruiz-Sautua, A.A.D. Barrio, and J.M. Mendías. Subword switching activity minimization to optimize dynamic power consumption. *IEEE Design & Test of Computers*, 26:68–77, 2009.

[Mue99] S.M. Mueller. On the scheduling of variable latency functional units. In *Symposium on Parallel Algorithms and Architectures*, pages 148–154, 1999.

[NRV03] R. K. Namballa, N. Ranganathan, and M. Varanasi. *CHESS: a tool for CDFG extraction and HLS of VLSI systems.* Phd, College of Engineering, University of South Florida, 2003.

[OG86] A. Orailogulu and D. Gajski. Flow graph representation. In *Proceedings of the 23rd Design Automation Conference*, pages 503–509, 1986.

[Pez71] S.D. Pezaris. A 40ns 17-bit by 17-bit array multiplier. *IEEE Transactions on Computers*, 20:442–447, 1971.

[PK89] P.G. Paulin and J.P. Knight. Force-directed scheduling for the behavioral synthesis of asic's. *IEEE Transactions on Computer-Aided Design*, 8:661–678, 1989.

[PK91] I.-C. Park and C.-M. Kyung. Fast and near optimal scheduling in automatic data path synthesis. In *Proceedings of the 28th ACM/IEEE Design Automation Conference*, pages 680–685, 1991.

[RRL00] V. Raghunatan, S. Ravi, and G. Lakshminarayana. Integrating variable-latency components into high-level synthesis. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 19:1105–1117, 2000.

[RSMMH05] R. Ruiz-Sautua, M.C. Molina, J.M. Mendías, and R. Hermida. Behavioural transformation to improve circuit performance in high-level synthesis. In *Proceedings of the Design, Automation and Test in Europe*, pages 1252–1257, 2005.

[SC97] D. Shin and K. Choi. Low power high level synthesis by increasing data correlation. In *Proceedings of the 23rd International Symposium of Low Power Electronics and Design*, pages 62–67, 1997.

[Skl60] J. Sklansky. Conditional-sum addition logic. *IRE Transactions on Electronics Computers*, 9:226–231, 1960.

[Sno78] E. A. Snow. *Automation of module set independent register-transfer level design.* Phd, Carnegie-Mellon University, 1978.

[TC90] The International Telegraph and Telephone Consultative Committee. 40, 32, 24, 16 kbits/s adaptative differential pulse code modulation g.726 recommendation. Technical report, International Telecommunication Union, 1990.

[Tom67] R.M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11:25–33, 1967.

[Tri87] H. Trickey. Flamel: a high-level hardware compiler. *IEEE Transactions on Computer-Aided Design*, 6:259–269, 1987.

[TS86] C.J. Tseng and D.P. Siewiorek. Automated synthesis of data paths in digital systems. *IEEE Transactions on Computer-Aided Design*, 5:379–395, 1986.

[Tya93] A. Tyagii. A reduced-area scheme for carry-select adders. *IEEE Transactions on Computers*, 42:1163–1170, 1993.

[VBI08] A. K. Verma, P. Brisk, and P. Ienne. Variable latency speculative addition: A new paradigm for arithmetic circuit design. In *Proceedings Of Design, Automation and Test in Europe*, pages 1250–1255, 2008.

[Wal64] C.S. Wallace. A suggestion for a fast multiplier. *IEEE Transactions on Electronic Computers*, 34:14–17, 1964.

[WC95] R.A. Walker and S. Chaudhuri. Introduction to the scheduling problem. *IEEE Design & Test of Computers*, 12:60–69, 1995.

[WDH01] W.F. Wallace, S.S. Dlay, and O.R. Hinton. Probabilistic carry state estimate for improved asynchronous adder performance. *IEE Proc. Computers and Digital Techniques*, 148:221–226, 2001.

[WM95] Wm. A. Wulf and Sally A McKee. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 23:20–24, 1995.

[WMGB95] T.C. Wilson, N. Mukherjee, M.K. Garg, and D. K. Banerji. An ilp solution for optimum scheduling, module and register allocation, and operation binding in datapath synthesis. *VLSI Design*, 3:21–36, 1995.

[WWB+02] Q. Wu, Y. Wang, J. Bian, W. Wu, and H. Xue. A hierarchical cdfg as intermediate representation for hardware/software codesign. In *Proceedings of the IEEE International Conference on Communications, Circuits and Systems and West Sino Expositions*, pages 1429–1432, 2002.

[ZG99] J. Zhu and D. Gajski. Soft scheduling in high level synthesis. In *Proceedings of the 36th Design Automation Conference*, pages 219–224, 1999.

Y ahora, caballeros – prosiguió, con una sonrisa amable –,
hemos llegado al final de nuestro pequeño misterio.
Ya pueden hacerme ustedes todas las preguntas que gusten,
sin miedo de que me niegue a responderlas.

Sherlock Holmes
Estudio en rojo
Arthur Conan Doyle

Y ahora, mi querido Watson, permítame decirle que llevamos
varias semanas trabajando con mucha intensidad y que, por una vez,
no estaría de más que nos ocupáramos de cosas mas placenteras.

Sherlock Holmes
El sabueso de los Baskerville
Arthur Conan Doyle